SignalR on .NET 6
The Complete Guide

The easiest way to enable real time two-way HTTP communication on .NET 6

Fiodar Sazanavets

SignalR on .NET 6 - the Complete Guide

The easiest way to enable real-time two-way HTTP communication on .NET 6

Fiodar Sazanavets

This book is for sale at http://leanpub.com/signalronnet6-thecompleteguide

This version was published on 2022-05-15

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Fiodar Sazanavets by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Here is the complete guide on using SignalR on .NET 6!

The suggested hashtag for this book is #signalr.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#signalr

Contents

1 - Introduction to SignalR **1**
What makes SignalR so great 2
Example use cases for SignalR 3
Who is this book for 4
The scope of this book 4
Prerequisites 5
How to use this book 5
Book structure 5
About the author 8
Getting touch with the author 8

2 - Setting up your project **9**
Prerequisites 9
Setting up your environment 9
Setting up SignalR hub 14
Making SignalR hub strongly-typed 17
Summary 18
Test yourself 18
Further reading 19

3 - In-browser SignalR clients **20**
Prerequisites 20
Setting up JavaScript client 21
Setting up Blazor WebAssembly client 27
Summary 33
Test yourself 34
Further reading 35

4 - External SignalR clients **36**
Prerequisites 36
Setting up .NET client 37
Setting up Java client 40
Setting up a raw WebSocket client 44
Summary 48

Test yourself . . . 49
Further reading . . . 49

5 - Sending messages to individual clients or groups of clients . . . 50
Prerequisites . . . 50
Broadcasting messages to all clients . . . 51
Sending messages to specific clients . . . 54
Working with client groups . . . 60
Summary . . . 65
Test yourself . . . 65
Further reading . . . 66

6 - Streaming in SignalR . . . 67
Prerequisites . . . 67
What is streaming used for . . . 68
Client streaming in SignalR . . . 69
Server streaming in SignalR . . . 74
Summary . . . 79
Test yourself . . . 80
Further reading . . . 80

7 - Advanced SignalR configuration . . . 81
Prerequisites . . . 81
Configuring SignalR server . . . 82
Configuring SignalR client . . . 90
Pros and cons of MessagePack protocol . . . 98
Summary . . . 104
Test yourself . . . 104
Further reading . . . 105

8 - Securing your SignalR applications . . . 106
Prerequisites . . . 106
What is CORS and why it's important . . . 107
Setting up single sign-on provider . . . 108
Applying authentication in SignalR . . . 118
Applying authorization in SignalR . . . 129
Summary . . . 132
Test yourself . . . 133
Further reading . . . 133

9 - Scaling out SignalR application . . . 135
Prerequisites . . . 135
Setting up Redis backplane . . . 136
Running multiple hub instances via Redis backplane . . . 140

Using HubContext to send messages from outside SignalR hub . . . 144
Summary . . . 146
Test yourself . . . 147
Further reading . . . 147

10 - Introducing Azure SignalR Service . . . 148
Prerequisites . . . 148
Setting up Azure SignalR Service . . . 149
Adding Azure SignalR Service dependencies to your application . . . 150
Overview of Azure SignalR Service REST API . . . 152
Summary . . . 156
Test yourself . . . 156
Further reading . . . 157
Wrapping up . . . 157

Answers to self-assessment questions . . . 159
Chapter 2 . . . 159
Chapter 3 . . . 159
Chapter 4 . . . 159
Chapter 5 . . . 159
Chapter 6 . . . 159
Chapter 7 . . . 160
Chapter 8 . . . 160
Chapter 9 . . . 160
Chapter 10 . . . 160

1 - Introduction to SignalR

If you are a software developer who builds web, mobile or internet of things (IoT) applications, you will appreciate how important it is for your applications to have the ability to communicate with the server asynchronously and in real time. Also, you probably realize that the standard request-response model of communication isn't fully suitable for modern apps. Sometimes, your client application (whether it's a web page, a mobile app or a service on an IoT device) needs to be able to receive a real-time update from the server that was actually triggered by an event on the server and not by a client request.

We see this functionality everywhere. When you use a messenger app, you would expect to receive a message as soon as someone has sent you one. When you control your IoT devices, you expect them to respond to your commands as soon as you trigger them.

But the problem with such functionality is that the said functionality is not always easy to implement. You could still use the classic request-response model and just carry on sending requests to the server until you receive a right type of response. But this would be wasteful. And, if your client application has limited bandwidth to work with, you may not even be able to do it at all, as continuous requests to the server may end up using up the entire bandwidth really quickly. As well as all of this, your code to implement such a behavior would probably be way more complicated than it should be.

There is also an alternative - `WebSocket` protocol. This is a much better solution. The protocol was specifically designed to enable two-way communication between the client and the server. All you have to do is establish a connection between the two endpoints. And, as long as this connection is live, the messages can flow both ways. As well as sending any arbitrary message from the client to the server, you can also send messages from the server to the client. And, on top of this, WebSocket protocol is very efficient. Maintaining the connection doesn't use much bandwidth at all.

However, WebSocket doesn't come without its own problems. It's far from being the easiest thing to work with. Out of the box, WebSocket protocol uses raw data instead of human-readable abstractions. It transmits messages by using raw bytes. So, it's up to you to do all the assembling and disassembling of messages. It's also up to you to monitor the state of the connection. The WebSocket code you'll have to write will probably look similar to this, which is neither very intuitive nor easily readable:

```
private static async Task ReceiveAsync(ClientWebSocket ws)
{
    var buffer = new byte[4096];

    while (true)
    {
        var result = await ws.ReceiveAsync(new ArraySegment<byte>(buffer), CancellationToken.None);
        if (result.MessageType == WebSocketMessageType.Close)
        {
            await ws.CloseOutputAsync(WebSocketCloseStatus.NormalClosure, string.Empty, CancellationToken.None);
            break;
        }
        else
        {
            Console.WriteLine(Encoding.Default.GetString(Decode(buffer)));
            buffer = new byte[4096];
        }
    }
}

private static byte[] Decode(byte[] packet)
{
    var i = packet.Length - 1;
    while (i >= 0 && packet[i] == 0)
    {
        --i;
    }

    var temp = new byte[i + 1];
    Array.Copy(packet, temp, i + 1);
    return temp;
}
```

Figure 1.1 - WebSocket implementation example

But if you are .NET developer, you won't have to deal with any of this. Enabling efficient real-time two-way communication will almost be as easy as making classes inside a single application call each other's methods. And all of this was made possible with a help of a library called `SignalR`, which is one of the in-built libraries of AS.NET Core.

What makes SignalR so great

SignalR is a library that is incredibly easy to implement compared to the alternatives. Using this library is just as easy as writing remote procedure calls. You will have some endpoint methods on your server that are just bog-standard C# methods. And, as long as you specify the methods of the same names on the client and put the expected parameters into them, the connection will be made and the method will be triggered.

Same applies the other way round. Your client will have event listeners with arbitrary names. And, as long as you spell the name of the listener correctly in your server-side code and apply expected data types as parameters, the listener on the client will be triggered.

For example, your server-side method may look like this:

```
public async Task BroadcastMessage(string message)
{
    await Clients.All.ReceiveMessage(message);
}
```

Figure 1.2 - Server-side SignalR method

And your client code that will trigger this method may look like this:

```
var message = $('#broadcast').val();
connection.invoke("BroadcastMessage", message)
    .catch(err => console.error(err.toString()));
```

Figure 1.3 - Client-side SignalR method invocation

Under the hood, SignalR library uses WebSocket protocol. But you, as a developer, don't have to worry about implementation details of it. Those are abstracted away to make things as convenient for you as possible.

But WebSocket is not the only protocol that SignalR uses, even though it is the default protocol it will try to use. It just happens that, although there aren't many use cases where you won't be able to use WebSocket, occasionally you may encounter such a situation. And this is why SignalR comes with two fallback protocols, which are server-sent events and long polling. The fallback order is as follows:

1. If possible, use WebSocket
2. If WebSocket can't be used, use server-sent events
3. If server-sent events can't be used, use long polling

Long polling is the least efficient protocol of them all. This is where the client sends a standard HTTP request to the server and just waits until the response is sent back. This is why you won't be using this protocol until absolutely necessary. But the main benefit of it is that absolutely any system that supports HTTP supports long polling.

If you need to, you can even explicitly specify the protocol in the client configuration. But in most cases, you won't have to. But even if you do, the choice of the protocol will have zero impact on the way you write your code. All your code will be identical regardless of the protocol being used.

Example use cases for SignalR

SignalR is a perfect library to be used in any scenarios where clients need to receive real-time updates from the server or there is high a high frequency of data exchange between the client and the server. As per the official documentation[1], the following are some of the examples where SignalR is an ideal library to use:

[1] https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-overview

- **High frequency data updates**: gaming, voting, polling, auction.
- **Dashboards and monitoring**: company dashboard, financial market data, instant sales update, multi-player game leader board, and IoT monitoring.
- **Chat**: live chat room, chat bot, on-line customer support, real-time shopping assistant, messenger, in-game chat, and so on.
- **Real-time location on map**: logistic tracking, delivery status tracking, transportation status updates, GPS apps.
- **Real time targeted ads**: personalized real time push ads and offers, interactive ads.
- **Collaborative apps**: coauthoring, whiteboard apps and team meeting software.
- **Push notifications**: social network, email, game, travel alert.
- **Real-time broadcasting**: live audio/video broadcasting, live captioning, translating, events/news broadcasting.
- **IoT and connected devices**: real-time IoT metrics, remote control, real-time status, and location tracking.
- **Automation**: real-time trigger from upstream events.

Who is this book for

This book will be useful to any ASP.NET Core developer who is interested in enabling real-time two-way communication between the clients and the server. Whether you are building a real-time chat application, a messenger app, or an IoT control hub, you will find the information in this book useful.

The scope of this book

This book will teach you everything you will need to know about SignalR, so you will be able to use it in any type of project where it can provide benefits. We won't go too deep into the inner workings of the library, but we will cover enough so you know how to use it in the most optimal way. We will even cover some use cases that aren't officially documented, but that may actually happen. For example, you will learn how to connect a raw WebSocket client to the SignalR server hub. This is a use case I have personally dealt with in one of my projects.

Another topic that we will address is how to integrate the latest .NET 6 and C# 10 features with SignalR. There is a wealth of information online on how to use SignalR, but since .NET 6 is relatively recent, there isn't much information on how to take advantage of the latest .NET 6 features while using it. And this book is aiming to address this gap.

In this book, you will be working on the same .NET solution throughout the chapters, adding new features to it as you go along. All the code samples used in the book are available in this GitHub repo:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide

Prerequisites

This book assumes that you are already somewhat familiar with .NET in general and ASP.NET Core in particular. Although anyone will be able to follow the examples provided in the chapters, to fully understand them, you need to understand what ASP.NET Core applications are and how they are structured.

One of the best places to learn ASP.NET Core is its official documentation website[2]. In our examples, we will mostly be using MVC (model-view-controller) template. So, to take the maximum benefit from this book, you will need to know what it is and how it works.

How to use this book

If you are completely new to SignalR, then my recommendation would be to follow this book from the beginning. However, you don't necessary have to read every chapter.

This book is intended to be both a complete tutorial and a reference book. So you can just read any specific chapter related to a specific SignalR feature that you are interested in.

Even though the book is structured in such a way that we add features to the same application throughout the chapters, you don't have to go through all previous chapters if you are interested only in a specific chapter. You can download the complete GitHub repository and just open the code sample folder that was relevant to the final part of the previous chapter.

Book structure

The book will consist of the following chapters

1 - Introduction to SignalR

This chapter provides a high-level overview of SignalR and outlines the structure for the remainder of the book.

2 - Setting up your project

This chapter will show you how to set up your environment, create an ASP.NET Core application and add a SignalR hub to it. We will cover how server-side components of SignalR work.

The chapter consists of the following topics:

- Setting up your environment
- Setting up SignalR hub
- Making SignalR hub strongly-typed

[2] https://docs.microsoft.com/en-us/aspnet/core/

3 - In-browser SignalR clients

In this chapter, you will learn how to set up in-browser SignalR clients. This is perhaps the most commonly used type of SignalR clients. For example, this is how you can build a real-time chat application.

The chapter consists of the following topics:

- Setting up JavaScript client
- Setting up Blazor WebAssembly client

4 - External SignalR clients

In this chapter, you will learn how to set up external self-contained applications as SignalR clients. For example, such a client may be an IoT application. We will cover all remaining types of clients that are officially supported and documented. But, on top of this, you will learn how to connect a raw WebSocket to the SignalR server, so you can write a client for it in absolutely any language of your choice.

The chapter consists of the following topics:

- Setting up .NET client
- Setting up Java client
- Setting up a raw WebSocket client

5 - Sending messages to individual clients or groups of clients

This chapter will walk you through the process of grouping SignalR clients. We will also have a look at how clients can be called selectively by the server. This is useful, because you won't always have to notify all connected clients when some event occurs.

The chapter consists of the following topics:

- Broadcasting messages to all clients
- Sending messages to specific clients
- Working with client groups

6 - Streaming in SignalR

As well as having an ability to make single calls, SignalR has an ability to stream data from the server to the client and vice versa. In this chapter, you will learn how to enable it.

The chapter consists of the following topics:

- What is streaming used for
- Client streaming in SignalR
- Server streaming in SignalR

7 - Advanced SignalR configuration

In this chapter, you will learn how to apply advanced configuration to SignalR, both client- and server-side. We will also cover an additional messaging protocol that can be used with SignalR - MessagePack. This protocol will allow you to substantially increase the performance of your communication.

The chapter consists of the following topics:

- Configuring SignalR server
- Configuring SignalR client
- Pros and cons of MessagePack protocol

8 - Securing your SignalR applications

Security is an important part of any software applications. You don't want unauthorized users to gain access to sensitive information or functionality. And SignalR is not exception. This chapter will teach you how to secure your SignalR application.

The chapter consists of the following topics:

- What is CORS and why it's important
- Setting up single sign-on provider
- Applying authentication in SignalR
- Applying authorization in SignalR

9 - Scaling out SignalR application

If your application is intended to work with a large number of clients, you will eventually need to scale it out. And this chapter will teach you how to scale out your SignalR hub.

The chapter consists of the following topics:

- Setting up Redis cache
- Running multiple hub instances via Redis backplane
- Using HubContext to send messages from outside SignalR hub

10 - Introducing Azure SignalR Service

You can scale out a SignalR server on premises, but you can also outsource the task to Azure cloud. This chapter will show you how to scale out your SignalR server by using Azure SignalR Service

The chapter consists of the following topics:

- Setting up Azure SignalR Service
- Adding Azure SignalR Service dependencies to your application
- Overview of Azure SignalR Service REST API

About the author

Fiodar Sazanavets is an experienced lead software engineer whose main area of expertise is Microsoft stack, which includes ASP.NET (Framework and Core), SQL Server, Azure, and various front-end technologies. Fiodar is familiar with industry-wide best practices, such as SOLID principles, software design patterns, automation testing principles (BDD and TDD) and microservices architecture.

Fiodar has built his software engineering experience while working in a variety of industries, including water engineering, financial, retail, railway and defence. He has played a leading role in various projects and, as well as writing software, he gained substantial experience in architecture and design.

Fiodar is an author of a number of technical books and online courses. He regularly writes about software development on his personal website, https://scientificprogrammer.net. He is also available to book for personal mentoring sessions via https://mentorcruise.com/mentor/fiodarsazanavets.

Getting touch with the author

If you want to get in touch with me regarding the content of the book, you can contact me via either Twitter or LinkedIn. Perhaps, there is additional content that you want to have included in the next edition of the book. Or maybe you found some errors in the current edition. Any feedback is welcome. And this is how you can get in touch:

Twitter: https://twitter.com/FSazanavets

LinkedIn: https://www.linkedin.com/in/fiodar-sazanavets/

2 - Setting up your project

This chapter will show you how to set up web application project and add server-side SignalR dependencies to it. We will cover the most fundamental components of SignalR and explain how the library works.

The chapter consists of the following topics:

- Setting up your environment
- Setting up SignalR hub
- Making SignalR hub strongly-typed

By the end of the chapter, you will have learned how to enable SignalR in an ASP.NET Core application. We will focus on an MVC application. However, the principles taught in this chapter will be universally applicable to other types of ASP.NET Core applications, including Razor Pages and Web API.

Prerequisites

To take the most out of this chapter, you will need to already be somewhat familiar with the structure of ASP.NET Core applications. There are no other prerequisites, as full instruction on how to set up your environment will be provided.

Another prerequisite is that you need a computer with either Windows, Mac OS, or Linux operating system. You will be working with .NET 6, which was designed to be compatible with any of these platforms.

The complete code samples from this chapter are available from the following location in the GitHub, which has separate folders corresponding to individual parts of the chapter:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide/tree/main/Chapter-02

Setting up your environment

The two main things that you will need in order to be able to follow the examples from this book are software development kit (SDK) for .NET 6 and a code editor that is compatible with C#.

If you already have these, you can skip this section. Otherwise, this is how you set everything up.

Setting up .NET 6 SDK

.NET 6 SDK contains both the platform that you will run your applications on and the collection of tools that will allow you to develop the applications. The best place to download the SDK from is its official website[3].

.NET 6 SDK will give you access to all the tools that you will need to instantiate your projects and build your applications. One of the powerful tools it comes with is `dotnet` command line interface (CLI) tool. It allows you to instantiate, build and run .NET applications via any command line terminal on any supported operating system. And all of the commands will be the same regardless of which operating system are you using.

Of course, you don't necessarily have to use CLI to build and run your application. You can do all of this via graphical user interface (GUI) of the integrated development environment (IDE) of your choice. However, because different operating systems have different IDEs available and they look completely different, all instructions provided in the book will be based on CLI commands.

Setting up a code editor

There is a significant difference between a code editor and an IDE. Code editor merely allows you to write the code. It provides all the necessary syntax highlighting and auto-formatting, but this is about it. You will need to either install special plug-ins or use external tools to be able to instantiate your projects or build your applications. But the biggest advantage that code editors have over IDEs is that code editors are much more light-weight and are much quicker to load.

IDE, on the other hand, is one-stop-shop for software development. All steps of developing your software can be done inside the IDE without having to rely on any external tools at all. Each of the actions, such as instantiating your project, running unit tests, building and running your application, would have a corresponding menu item in the GUI. But all these features come at a price. IDEs would occupy much more disk space than code editors and they are much more resource-hungry.

Whether you will decide to use a code editor or an IDE, it doesn't matter. All examples provided in this book would work with either.

Now, let's set up the tools specific to the OS you intend to use. You don't have to read each of this sections. Just pick the one that is relevant to you.

Setting up on Windows environment

If you are using Windows, here are the options that are available to you.

Visual Studio 2022

Perhaps the most popular .NET IDE for Windows is Visual Studio. In order to be able to work with .NET 6, you will need Visual Studio 2022. You can download it from its official page[4].

[3] https://dotnet.microsoft.com/en-us/download/dotnet/6.0
[4] https://visualstudio.microsoft.com/downloads/

There are three editions of Visual Studio 2022: Community, Professional and Enterprise. The community edition is free to download, but you will need to register.

JetBrains Rider

Rider IDE by JetBrains doesn't come for free. But there is a reason for it. It contains many useful tools, such as ReSharper, that allows you to decompile .NET assemblies back into human-readable code.

But even though the IDE itself isn't free, there is a free trial available. It can be downloaded from its official page[5].

Visual Studio Code

Even though this tool sounds like Visual Studio, it's a completely different tool. Visual Studio Code (also known as VS Code) is a lightweight, but powerful, code editor. It comes completely free of charge and you won't need to register to use it. Also, it's open-source and ever-green, which means that it will continuously receive updates, so you won't have to download a new version every time a new SDK becomes available. On top of all of that, it has a very powerful plugin system, so you can almost turn it into a fully-fledged IDE.

VS Code is available on any platform. And you can download it from its official website[6].

Setting up on Mac OS environment

The following options are recommended if you are a Mac user.

Visual Studio for Mac

Visual Studio for Mac is a Mac version of the classic Windows-based Visual Studio IDE. You can download it from its official page[7]. It is available to download free of charge, but you will have to register.

JetBrains Rider

Rider IDE by JetBrains doesn't come for free. But there is a reason for it. It contains many useful tools, such as ReSharper, that allows you to decompile .NET assemblies back into human-readable code.

But even though the IDE itself isn't free, there is a free trial available. It can be downloaded from its official page[8].

[5] https://www.jetbrains.com/rider/
[6] https://code.visualstudio.com/
[7] https://visualstudio.microsoft.com/downloads/
[8] https://www.jetbrains.com/rider/

Visual Studio Code

Even though this tool sounds like Visual Studio, it's a completely different tool. Visual Studio Code (also known as VS Code) is a lightweight, but powerful, code editor. It comes completely free of charge and you won't need to register to use it. Also, it's open-source and ever-green, which means that it will continuously receive updates, so you won't have to download a new version every time a new SDK becomes available. On top of all of that, it has a very powerful plugin system, so you can almost turn it into a fully-fledged IDE.

VS Code is available on any platform. And you can download it from its official website[9].

Setting up on Linux environment

Linux has fewer options available than either Windows or Mac OS. However, you will still be able to set up your development environment with the full set of capabilities.

JetBrains Rider

Rider IDE by JetBrains doesn't come for free. But there is a reason for it. It contains many useful tools, such as ReSharper, that allows you to decompile .NET assemblies back into human-readable code.

But even though the IDE itself isn't free, there is a free trial available. It can be downloaded from its official page[10].

Visual Studio Code

Even though this tool sounds like Visual Studio, it's a completely different tool. Visual Studio Code (also known as VS Code) is a lightweight, but powerful, code editor. It comes completely free of charge and you won't need to register to use it. Also, it's open-source and ever-green, which means that it will continuously receive updates, so you won't have to download a new version every time a new SDK becomes available. On top of all of that, it has a very powerful plugin system, so you can almost turn it into a fully-fledged IDE.

VS Code is available on any platform. And you can download it from its official website[11].

Enabling development HTTPS certificate

This step is not strictly necessary, but it will allowed you to add HTTPS protocol to your applications via a self-signed development certificate. .NET SDK comes with its own self-signed certificate and to ensure that it works, you need to trust it. To do so on either Windows or Mac OS, all you have to do is run the following command:

[9] https://code.visualstudio.com/
[10] https://www.jetbrains.com/rider/
[11] https://code.visualstudio.com/

```
dotnet dev-certs https --trust
```

On Linux, the process of trusting the certificate would be different, so you would need to read the manual specific to the distro you are using. However, some guidance will be provided in **further reading** section.

Or you may choose not to use HTTPS at all. It's up to you, as it's not strictly needed to work with the code samples from the book. But please use it if you can, as it will make your application slightly more similar to what you would build in a real-life project.

Setting up the solution

In your file system, create a folder and call it `LearningSignalR`. Now, open any command line terminal of your choice. It doesn't matter whether it's PowerShell, cmd, bash or anything else that your OS has. The commands we will execute will work on any of them.

In the terminal, make sure you navigate to the newly created `LearningSignalR` folder and execute the following command inside of it:

```
dotnet new sln
```

This command is expected to generate a solution file with the same name as the folder. So, in our case, to ensure that the command has worked, we need to check whether our folder contains `LearningSignalR.sln` file.

Next, we will instantiate a project based on ASP.NET Core MVC template. To do so, while having the terminal open in the solution folder, we will excute the following command:

```
1 dotnet new mvc -o SignalRServer
```

This command will generate `SignalRServer` folder inside the solution folder with all the default code setup for ASP.NET Core MVC project. We will now need to add the project to the solution. And to do so, this is the command we will need to execute:

```
1 dotnet sln add SignalRServer/SignalRServer.csproj
```

The right-most argument of this command is the relative path to the C# project file, which has `csproj` extension.

That's it. We now have a solution with an MVC project in it. If you are using an IDE, you can open the solution by double-clicking on the `sln` file. Otherwise, you can use your code editor to open the solution folder.

Now, we are ready to add SignalR components to our project.

Setting up SignalR hub

The server-side SignalR components center around the so-called **hub**. SignalR hub is an equivalent to MVC or Web API controller. Or, if you are familiar with gRPC, it is an equivalent of gRPC service implementation. It is a class with a collection of methods that SignalR clients will be able to remotely trigger.

We will add a basic example of a hub to our project and register all necessary dependencies. Then, we will go through its structure in a little bit more detail.

Adding SignalR hub to the project

We will follow similar conventions to MVC. As we have **Models**, **Views** and **Controllers** folders inside `SignalRServer` project, we will add another folder and call it **Hubs**. We will then create `LearningHub.cs` file inside this folder and populate it with the following content:

```
using Microsoft.AspNetCore.SignalR;

namespace SignalRServer.Hubs
{
    public class LearningHub : Hub
    {
        public async Task BroadcastMessage(string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", message);
        }

        public override async Task OnConnectedAsync()
        {
            await base.OnConnectedAsync();
        }

        public override async Task OnDisconnectedAsync(Exception? exception)
        {
            await base.OnDisconnectedAsync(exception);
        }
    }
}
```

So, let's go through the structure of the class. This is a basic example, so don't worry. You won't be overwhelmed with the information.

SignalR hub overview

Firstly, a class that we want to use as a SignalR hub needs to inherit from `Hub` class, which is a part of `Microsoft.AspNetCore.SignalR` namespace. This namespace is included in the standard ASP.NET Core libraries, so we won't have to add any external references. But we still need to reference this namespace somewhere. We can either do it in the file containing the class, like we have done in the above example, or you can just add the `using` statement anywhere in your application and prepend `global` keyword to it to make it available everywhere within the project. There is no difference between these two ways of referencing namespaces. You can choose either, depending on your personal preferences or any specific guidelines that you follow.

Inside the hub, we have `BroadcastMessage` method that accepts a `string` parameter. This is an example of a method that clients will be able to call once they are connected to the hub. There is nothing special about this method, other than it needs to return a `Task`. It can have any parameters of any data types. The messages are sent and received in the form of JSON, so any data that gets transferred can be easily deserialized either into basic data types, like `string` or `int` or more complex types, like `class` or `struct`.

There is a limit to how many parameters you can have. But the number is reasonably large, so, as long as you are writing clean code and following best practices, you shouldn't worry about running out of available parameters.

So, `BroadcastMessage` method receive a `string` message from a client. Then, inside the method, this message gets re-sent to all clients, including the one that has sent it. Sending message to any entity that has tuned in is known as broadcasting. And this is precisely why the method is called `BroadcastMessage`.

But how does the hub know what clients are connected to it? Well, that's easy. The base `Hub` class has a property called `Clients`. This property has the details of all connected clients. And you can use this property to choose which clients to send the message to.

In our case, to make things as simple as possible, we are choosing to send the message to all clients. We do so by accessing `All` property. But there are multiple different ways of selecting specific clients to send the message to. We will cover this in **chapter 5**.

To actually send the message and trigger a specific event in the client code, we call `SendAsync` method. The first parameter of this method is the even name in the client code. The spelling has to be exactly the same as it's spelled on the client. Otherwise the event won't be triggered. The other parameters are the input parameters for the event.

Next, we have overrides of `OnConnectedAsync` and `OnDisconnectedAsync` methods from the original `Hub` class. In our example, we aren't doing anything with them. But these are the methods that get triggered when a client establishes connection or disconnects. Because disconnection can happen due to an error, `OnDisconnectedAsync` methods also has nullable `Exception` parameter.

There are also some other public members of `Hub` class that we haven't covered. There is `Groups` property, which allows you to assign individual clients to groups and then broadcast messages to

the members of a particular group. There is also `Context` property, which contains information of the current session. For example, you can extract a unique client identifier from this property, which gets auto-generated when the client connects and remains constant until the client disconnects. `Context` property is conceptually similar to `HttpContext` from Web API controllers. We will cover both `Groups` and `Context` in **chapter 5**.

One important thing to remember about a SignalR hub is that its lifetime is restricted to a single call. So don't store any durable data in it. It will all disappear once the next call is executed.

And this completes a basic overview of SignalR hub. Now, we need to enable it, so our clients can actually access it.

Enabling SignalR hub endpoint

Enabling SignalR hub is simple. All we have to do is add a couple of lines to our `Program.cs` file. First, we will need to reference the hub namespace by adding this `using` statement to the file:

```
1  using SignalRServer.Hubs;
```

You can prepend it with `global` keyword to make it universally accessible to all files, so you won't have to insert the `using` statement again.

Next, we will need to add the following line just before `app` variable gets instantiated:

```
1  builder.Services.AddSignalR();
```

This statement enables us to use SignalR middleware in the application. Finally, we need to add the following line before `Run` method is called on the `app` variable:

```
1  app.MapHub<LearningHub>("/learningHub");
```

In this line, we have mapped our SignalR hub to a specific path in the URL. So, a client is now able to register with the hub by submitting a HTTP request to `{Base URL}/learningHub` address. The initial request is done via the standard HTTP protocol. But then, if the protocol needs to change (for example, to WS, which represents WebSocket), this will happen under the hood. You, as a developer, won't have to worry about it.

That's it. Our hub is now registered and our SignalR server is ready to accept client connections. But there is one additional modification that we can apply to the hub to minimize the risk of misspelling the client event names. And this is what we will do next.

Making SignalR hub strongly-typed

So far, when we've been calling clients from the hub, we have used just a normal `string` to specify the client-side event name that needs to be triggered. But the problem with using a `string` is that it can contain any arbitrary text. If we misspell the name, the code will not warn us. Perhaps, it's not a big problem in our case, because we only have one such a call, so it's easy enough to manage. But what if we had a complex hub (or multiple hubs) with many of such calls?

Fortunately, there is a solution. You can make your hub strongly-typed, so any method you have on the client can be transformed into a C# method representation. And you can't misspell a name of a C# method. Your IDE will show an error and your code won't compile.

To do so, we will need to add an interface that represent client events. And then we will enforce this interface on the hub. We will start by adding `ILearningHubClient.cs` file to the root folder of our project. This file will contain the following content:

```
namespace SignalRServer
{
    public interface ILearningHubClient
    {
        Task ReceiveMessage(string message);
    }
}
```

So, as you can see, we have a `Task` method with the name of `ReceiveMessage` - the same name that we previously specified as the client-side event name in the call to `SendAsync` method. Now, we will modify our hub. To make it simple, we will replace the content of `LearningHub.cs` file with the following:

```
using Microsoft.AspNetCore.SignalR;

namespace SignalRServer.Hubs
{
    public class LearningHub : Hub<ILearningHubClient>
    {
        public async Task BroadcastMessage(string message)
        {
            await Clients.All.ReceiveMessage(message);
        }

        public override async Task OnConnectedAsync()
        {
            await base.OnConnectedAsync();
```

```
15          }
16
17          public override async Task OnDisconnectedAsync(Exception? exception)
18          {
19              await base.OnDisconnectedAsync(exception);
20          }
21      }
22  }
```

Let's break it down. Our class no longer just inherits from `Hub` base class. We also have our interface name in the angle brackets next to the base class name. And that forces us to not be able to use `SendAsync` method on the objects that represent clients. Instead, we have to use any of the methods defined in the interface. So now, we can no longer accidentally misspell the name.

And that concludes the overview of a SignalR hub. We are now ready to start connecting clients to it. But first, let's summarize what we have covered.

Summary

SignalR hub is a server-side class that inherits from `Hub` class of `Microsoft.AspNetCore.SignalR` namespace. This is the class that would have any arbitrary methods that the clients will be able to trigger remotely.

As well as having arbitrary methods, SignalR hub can override the default connection and disconnection events. The disconnection even accepts a nullable `Exception` parameter, because a disconnection may happen due to a failure. Other than that, a hub class would have access to `Clients`, `Groups` and `Context` properties.

SignalR hub is not durable, so no persistent data can be stored in its variables. The data will be deleted before the next call. To enable SignalR hub, you need to enable SignalR middleware in `Program.cs` file and then map a specific hub to a specific URL path.

SignalR hub can be strongly typed. This is achieved by creating an interface containing allowed client methods and then forcing the client-related properties of the hub to implement it.

In the the next chapter, we will start connecting clients to our hub. We will focus on in-browser clients first.

Test yourself

1. What is a SignalR hub?
   A. A cloud provider hosting SignalR service
   B. The application that hosts SignalR components

   C. A class that contains the methods that clients can trigger
   D. The machine that hosts SignalR application
2. Which of the following properties exists in the `Hub` base class?
   A. Context
   B. Clients
   C. Groups
   D. All of the above
3. What is the best way to minimize the risk of misspelling client-side event name on SignalR server?
   A. Save it in a `const` variable
   B. Save it in a `readonly` variable
   C. Enforce client interface on the hub
   D. Have a unit test to check the spelling

Further reading

Official SignalR Hub documentation: https://docs.microsoft.com/en-us/aspnet/core/signalr/hubs

3 - In-browser SignalR clients

SignalR Hub is only useful if you have clients connected to it. And in this chapter, you will learn how to connect in-browser clients.

We will start with in-browser clients because they are slightly easier to set up than external clients. For example, you won't have to create external applications and you will need to apply slightly less configuration to make them work. For example, since your client will already reside in the same web application that hosts your SignalR hub, you won't have to explicitly specify the base URL.

These types of SignalR clients are useful when you need to build a web application with real-time update capabilities. It could be a chat application, where you instantly see the messages sent to you. It could be auto-updating news feed. Or it could be some dashboard with real-time metrics.

But front-end technologies are no longer limited to HTML, JavaScript and CSS. These days, you can also run compiled code in browsers, which became possible with the invention of WebAssembly. And .NET happens to have its own tool to write code that gets compiled to WebAssembly. This tool is known as Blazor. And, in this chapter, as well as building a standard JavaScript client, we will build a Blazor client too.

The chapter consists of the following topics:

- Setting up JavaScript client
- Setting up Blazor WebAssembly client

By the end of this chapter, you will have learned how to call your SignalR hub from the code running in browser. You will have learned different ways of adding SignalR client dependencies to different technology types and then you will have learned how to get your client to establish a persistent connection with your server-side hub.

Prerequisites

This chapter assumes that you already have set up your development environment, as described in **chapter 1**. You will need the following:

- A machine with either Windows, Mac OS or Linux operating system
- A suitable IDE or code editor (Visual Studio, JetBrains Rider or VS Code)
- .NET 6 SDK (or newer)

Also, since we are continuing to build on top of the application that we have developed in the previous chapter, we need the code that we have written previously. If you have skipped the previous chapter, you can access the complete code from the following location in the GitHub repository:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide/tree/main/Chapter-02/Part-03/LearningSignalR

The complete code samples from this chapter are available from the following location in the GitHub repo, which has separate folders corresponding to individual parts of the chapter:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide/tree/main/Chapter-03

Setting up JavaScript client

The first (and the simplest) SignalR client we will set up is JavaScript client. And even though you can use SignalR client JavaScript library from a stand-alone application, we will use it from our application's own pages. The client will run in the browser.

But if you later want to set up JavaScript SignalR client in a different type of application (including a stand-alone JavaScript app), the setup process will be very similar. And so will be the code. It is the same client library after all.

So, let's now go ahead and set up our client.

Adding SignalR client dependencies

There are two ways you can add SignalR library dependencies to the front-end. You can either instantiate it via NPM and copy JavaScript file into your application's file system, which you will then reference from your front-end HTML, or you can just obtain that file from a publicly accessible URL of a content delivery network (CDN) as a direct reference in your HTML. The former method is more involved, but once completed, this file will be a part of your application forever. The latter method is much easier, but because you are dealing with third-party URL that you have no control over, there is a very slight risk that the file might become unavailable.

But because such a risk is very low, we will have a look at using CDN reference first. Then, if you are still interested in getting your hands dirty and obtaining the library via NPM, this information will be provided too.

Adding SignalR library via CDN

CND is nothing more than a fancy name for a URL that contains the file with the content you are looking for. There are various websites where developers can publish JavaScript library as a file with `js` extension. And that file will be accessible from the server via a direct link.

And in our case, the best place to put the the reference to a file is via the `_Layout.cshtml` file, which is located in `Shared` folder inside `Views` folder. Just open that file and find the following line:

```
<script src="~/lib/jquery/dist/jquery.min.js"></script>
```

All you have to do is place the following line above it:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.1/signalr.\
min.js"></script>
```

That's it. You have now added a reference to the JavaScrip SignalR library. The only caveat that you need to remember is that this reference needs to be placed before the `script` element that has a reference to `js/site.js`. `site.js` is the file where we will be using the library. But we can't use it unless the SignalR client library loaded first. And because `script` elements load their respective resources in the order that they are placed in, we need to load the dependency before we load the code that uses the dependency.

Adding SignalR library via NPM

As I said earlier, this process is a bit more involved. First of all, you need to make sure that you have NPM installed on your machine. You can obtain it from its official website[12].

NPM stands for Node Package Manager. And originally it was developed as a system of adding external libraries to Node.js applications, which allow you to run JavaScrip directly on the host machine rather than just in a browser. This is why Node.js is primarily used to build applications where JavaScript represents both server-side and client-side code.

But since its inception, NPM became much more than just a library management system for Node.js. It appears to be a convenient way of obtaining external dependencies for client-side applications too. And this is why it became popular with front-end developers who don't even use Node.js. And this is what we will do too.

Once you have NPM installed and configured, you can open command line terminal inside `SignalRServer` project folder and execute the following command:

```
npm init -y
```

This will instantiate Node.js project inside the folder and create `package.json` and `package.lock.json` files. These files are Node.js equivalent of .NET project files with `.csproj` extension. And we will need them to download JavaScript libraries from NMP. To do so, execute the following command:

```
npm install @microsoft/signalr
```

[12] https://nodejs.org/en/download/

This will create a folder called `node_modules`. This folder will contain raw code of all the libraries that the original SignalR libraries was using. But we won't need to worry about any of that. We will simply need to open `node_modules` folder and then navigate to `@microsoft/signalr/dist/browser` folder. If you are using Windows, then you would need to use back slashes (\) rather than forward slashes (/) in the folder path. Or just navigate through the folders via the GUI file manager.

What you will need to do then is copy either `signalr.js` or `signalr.min.js` from this folder into `lib` folder that is located under `wwwroot` folder of your project. It doesn't matter which file you copy. They work exactly the same. The only difference between them is that `signalr.js` contains full human-readable code, while `signalr.min.js` is miniated version of it, where all unnecessary components (new lines, white spaces, etc.) have been removed and all local variable names were made as short as possible. Miniation is a standard practice with front-end resources, such as JavaScript and CSS files. As these resources are downloaded from the network, miniation is done to make the file size as small as possible. But functionally, those files will be identical to the original.

To follow a standard convention, you may also create `signalr` folder inside the `lib` folder and move the JavaScript file there. But this doesn't matter from the functionality perspective.

Now, we can insert the following line into `_Layout.cshtml` file to reference the SignalR library:

```
1  <script src="~/lib/signalr/signalr.js"></script>
```

Make sure this reference is placed before the `script` element that contains the reference to `site.js`. If you have copied `signalr.min.js` instead of `signalr.js` or if your file path looks different, then modify the content of the `src` attribute accordingly.

Our SignalR client library reference has been added. We can now delete `node_modules` folder along with `package.json` and `package.lock.json` files, as we will no longer need them. And we can start adding the actual client functionality.

Adding SignalR client logic

Our SignalR client will operate from a web page, so we will need to create a web page with controls that will allow us to trigger messages from the client. This will be the main page of our application, so we will apply all of our changes to `Index.cshtml` file, which is located under `Home` folder of the `Views` folder. We will simply replace the content of the file with the following:

```
@{
    ViewData["Title"] = "Home Page";
}

<div class="row" style="padding-top: 50px;">
    <div class="col-md-4">
        <div class="control-group">
            <div>
                <label for="broadcast">Message</label>
                <input type="text" id="broadcast" name="broadcast" />
            </div>
            <button id="btn-broadcast">Broadcast</button>
        </div>
    </div>

    <div class="col-md-7">
        <p>SignalR Messages:</p>
        <pre id="signalr-message-panel"></pre>
    </div>
</div>
```

We will need to apply some styling to the page to make it usable. To do so, open `site.css` file, which is located in `css` folder of `wwwroot`, and add the following content to it without modifying any existing content:

```
.body-content {
    padding-left: 15px;
    padding-right: 15px;
}

.control-group {
    padding-top: 50px;
}

label {
    width: 100px;
}

#signalr-message-panel {
    height: calc(100vh - 200px);
}
```

Finally, we will need to add JavaScript code that will get triggered when the buttons on the page

are pressed. We will simply insert the following content into `site.js` file, which is located inside `js` folder of `wwwroot`.

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/learningHub")
    .configureLogging(signalR.LogLevel.Information)
    .build();

connection.on("ReceiveMessage", (message) => {
    $('#signalr-message-panel').prepend($('<div />').text(message));
});

$('#btn-broadcast').click(function () {
    var message = $('#broadcast').val();
    connection.invoke("BroadcastMessage", message).catch(err => console.error(err.to\
String()));
});

async function start() {
    try {
        await connection.start();
        console.log('connected');
    } catch (err) {
        console.log(err);
        setTimeout(() => start(), 5000);
    }
};

connection.onclose(async () => {
    await start();
});

start();
```

Let's now go through this code step-by-step to see what it's doing.

Overview of JavaScript SignalR client implementation

In the above code, the first thing that we are doing is building an object that represents a SignalR connection. We are doing it via `HubConnectionBuilder` method of `signalR` object from the SignalR JavaScript library. We then specify the URL of the SignalR hub via `withUrl` method. The parameter could be either a relative path or a full URL. We are using relative path, because we are calling the

hub from the same application that hosts it. However, if you needed to establish a SignalR connection from an external app, then you would need to use the full URL. We set the default logging level on the `connection` object, which is optional. And then we build the object.

After this, we are adding an event listener to the connection object. The name of the event is `ReceiveMessage`, as we have previously defined in the hub. When the server-side code triggers this event, the message that came from it is prepended inside a panel on the screen, which is represented by a HTML element with the `id` attribute of `signalr-message-panel`.

We then associate a click event with the HTML button that has the `id` attribute of `btn-broadcast`. The message is read from an input box with the `id` attribute of `broadcast`. And then, by calling `invoke` method on the `connection` object, we trigger `BroadcastMessage` method in the server-side SignalR hub while passing the message as a parameter. We have also added optional error handler to this invocation.

Then we define `start` function, which will trigger the SignalR connection to start. There is some logic in it which will attempt to start the connection with retries if, for whatever reason, it couldn't have been started. Because we expect our SignalR connection to be live for as long as the page is open in the browser, we also associate this function with `onclose` event on the `connection` object. Basically, if the connection breaks at any point, it will automatically restart. Then, once we have all of our event handlers defined, we just call the `start` function to ensure that the connection is started automatically when the page has loaded.

We have now completed the setup of our JavaScript SignalR client. Now, we can launch our application and see it in action.

Launching JavaScript client

We need to build and run our application and then open its main page in the browser. There are multiple ways of doing it, depending on which IDE you are using. But one of the common and the simplest ways of doing it is to simply execute `dotnet run` command from inside `SignalRServer` project folder.

Once the console output indicates that the application is running, you can open it in the browser. The address that you will need to navigate to will be listed in `launchSettings.json` file inside the `Properties` folder of your project. Any URL listed under `applicationUrl` entry will be suitable.

On the home page of the application, you should see a text box and a button. When you enter some text and press the button, you should see the same message appearing under **SignalR Messages** title. And you can do it as many times as you want. And this demonstrates how SignalR hub on the server can trigger an appropriate event in your client code, as demonstrated by the screenshot below:

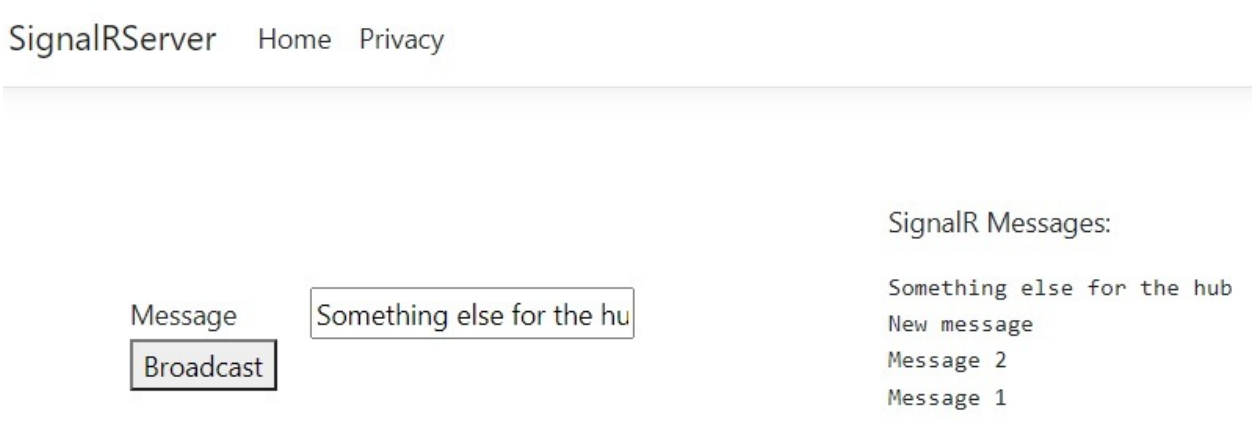

Figure 3.1 - JavaScript client demonstration

And this concludes our overview of a JavaScript SignalR client. It's very basic at this point, but we will add some more advanced functionality to it in the subsequent chapters. But for now, we will set up another browser-based SignalR client based on Blazor WebAssembly.

Setting up Blazor WebAssembly client

Blazor WebAssembly is a .NET implementation of WebAssembly technology. This technology allows you to run compiled code in browser without having any special plugins installed. It's a common standard that all browser manufacturers support in their products.

A detailed description of Blazor is beyond the scope of this book. But we will go into enough details of it to set up a SignalR client inside it. Also, an important thing to remember is that there are two distinct types of Blazor - WebAssembly and Blazor Server. The former type allows you to compile your code into WebAssembly and run it in browser, which is what we need. The latter type, despite having the same syntax, is set up differently. There's no compiled code on your web page. Instead, JavaScrip is generated from your code when you build your application. And then that JavaScript interacts with the code on your server. The only useful thing to know about Blazor Server in our context is that it actually uses SignalR for this communication.

So, let's now go ahead and create a Blazor WebAssembly application. We will do so by executing the following command inside the `LearningSignalR` solution folder:

```
1 dotnet new blazorwasm -o BlazorClient
```

We will then need to add it to our solution by executing the following command:

```
dotnet sln add BlazorClient\BlazorClient.csproj
```

Alternatively, you can just create the application from **Blazor WebAssembly** template via the GUI of your IDE. But just make sure that you don't select **hosted** model. Even though we will be hosting our WebAssembly application inside our main web application, the default hosted application template is not what we need. This template will generate the host application for you. But we already have one.

Now, once our application has been created, we will need to apply SignalR client references to it and add our client code.

Setting up SignalR client components

To enable SignalR client components in the Blazor application, we will first need to add a NuGet package that contains them. These components reside in `Microsoft.AspNetCore.SignalR.Client` NuGet package. And we can add it to our project by executing the following command from inside the `BlazorClient` project folder:

```
dotnet add package Microsoft.AspNetCore.SignalR.Client
```

And now we can start adding our client. The client logic will reside inside `Client.razor` file that we will create inside the `Pages` folder. The content of the file will be as follows:

```
@page "/client"
@using Microsoft.AspNetCore.SignalR.Client
@inject NavigationManager NavigationManager
@implements IAsyncDisposable

<h1>Blazor WebAssembly Client</h1>

<div class="row" style="padding-top: 50px;">
    <div class="col-md-4">
        <div class="control-group">
            <div>
                <label for="broadcastMsg">Message</label>
                <input @bind="message" type="text" id="broadcastMsg" name="broadcast\
Msg" />
            </div>
            <button @onclick="BroadcastMessage" disabled="@(!IsConnected)" >Broadcas\
t</button>
        </div>
    </div>

```

```
    <div class="col-md-7">
        <p>SignalR Messages:</p>
        <pre>
            @foreach (var message in messages)
            {
                @message<br/>
            }
        </pre>
    </div>
</div>
```

If you look at it carefully then you will see that it has almost identical markup to the page that uses JavaScript client. The only differences are related to syntax differences between the technologies. It's not pure HTML anymore. It's a combination between C#, HTML and bespoke Razor syntax, which allows you to write HTML helpers in an easy way.

At the beginning, we define the default path to the page containing this component by utilizing `@page` directive. In our case, the path is `/client`. Then we add references to any external namespaces via `@using` directive. The only one we care about is `Microsoft.AspNetCore.SignalR.Client`. Then we inject a dependency of `NavigationManager` by using `@inject` directive. We will need this object later to construct the URL to the SignalR hub endpoint. Finally, we use `@implements` directive to say that this component implements `IAsyncDisposable` interface.

Then, inside the HTML markup, we have some key words that begin with `@` character. All of those are related to C# code that we will insert shortly. `@bind` directive binds the content of an `input` element to a C# variable called `message`. Whenever you change the text, the content of the variable will change too. `@onclick` directive on the `button` element triggers a C# method called `BroadcastMessage`. And then there is some logic inside a `pre` element with the `id` attribute of `signalr-message-panel` that populates the element with the content of `messages` collection.

It will all make more sense when we add the actual C# code. So let's do it now. Just place this section at the bottom of the file:

```
@code {
    private HubConnection hubConnection;
    private List<string> messages = new List<string>();
    private string? message;

    protected override async Task OnInitializedAsync()
    {
        hubConnection = new HubConnectionBuilder()
            .WithUrl(NavigationManager.ToAbsoluteUri("/learningHub"))
            .Build();

```

```
12          hubConnection.On<string>("ReceiveMessage", (message) =>
13          {
14              messages.Add(message);
15              StateHasChanged();
16          });
17
18          await hubConnection.StartAsync();
19      }
20
21      private async Task BroadcastMessage() =>
22         await hubConnection.SendAsync("BroadcastMessage", message);
23
24      public bool IsConnected =>
25          hubConnection?.State == HubConnectionState.Connected;
26
27      public async ValueTask DisposeAsync()
28      {
29          await hubConnection.DisposeAsync();
30      }
31  }
```

Now, this is pure C#. But the principles are very similar to what we had in our JavaScript client. We are building `hubConnection` object from `HubConnectionBuilder` class. Then we register `ReceiveMessage` event that the server-side hub can trigger. In this event, we add the messages to the collection and call a Blazor-specific `StateHasChanged` method, which will trigger re-execution of the code defined in the markup. This will make sure that the panel with SignalR messages always gets populated with all messages we have ever received from the server, including the latest message. Then we start our hub.

`BroadcastMessage` is the method that get triggered when we press the button on the page. It will do so by calling `SendAsync` method on `hubConnection` object. This method accepts the name of the method on the SignalR hub, followed by a complete list of input parameters.

We have a read-only property of `IsConnected`, which ensures that appropriate controls on the page are only shown when the hub is actually connected. With C# SignalR client, the connection management will happen automatically and you won't have to add any additional logic to deal with occasional disconnections. Then, to make sure that the `hubConnection` object stops the connection and releases all unmanaged resources when we no longer need them, we dispose of it whenever we dispose of the Blazor object itself. This, for example, will happen when we close the page.

We now have SignalR client logic in our Blazor application. What we need to do now is set it up so it can be hosted in our main web application.

Hosting Blazor application inside an existing ASP.NET Core application

Our current Blazor application is set up as stand-alone self-hosted app, so we need to make some changes to be able to host it in another application. And the first change that we will do is open `Program.cs` file inside `BlazorClient` project and remove the following line:

```
builder.RootComponents.Add<App>("#app");
```

Next, we will need to add a `BlazorClient` project reference to the `SignalRServer` project. You can either do it via the GUI of your IDE, or manually add the following snippet to `SignalRServer.csproj` file:

```
<ItemGroup>
  <ProjectReference Include="..\BlazorClient\BlazorClient.csproj" />
</ItemGroup>
```

Next, we will need to install a NuGet package that will allow our `SignalRServer` application to act as a server that can host a Blazor WebAssembly application. The NuGet package is called `Microsoft.AspNetCore.Components.WebAssembly.Server` and you can install it by executing the following command inside `SignalRServer` project folder:

```
dotnet add package Microsoft.AspNetCore.Components.WebAssembly.Server
```

Next, we add a page that will display the Blazor component. We will open `HomeController.cs` file that's located inside `Controllers` folder and add the following method to it:

```
public IActionResult WebAssemblyClient()
{
    return View();
}
```

We will now need to add the corresponding view. To do so, add `WebAssemblyClient.cshtml` to the `Home` folder inside the `Views` folder. The content of this file will be as follows:

```
@{
    ViewData["Title"] = "Home Page";
}

@using BlazorClient.Pages;

<component type="typeof(Client)" render-mode="WebAssemblyPrerendered" />

<script src="_framework/blazor.webassembly.js"></script>
```

What we have done here is loaded the `Client` component from the `BlazorClient` assembly and loaded appropriate JavaScript file that will allow our WebAssembly component to work and interact with the controls on our page. `blazor.webassembly.js` file comes from the framework. We have already added the necessary framework components by installing `Microsoft.AspNetCore.Components.WebAssembly.Server` NuGet package. `WebAssemblyPrerendered` render mode is telling the platform to render the code as WebAssembly. If it cannot render it (for example, if you run it on an old or severely restricted browser), it will generate some JavaScript in the browser, but will host the actual component in on the server. It will, effectively, become equivalent to Blazor Server.

Next, we need to add a navigation link to the view we have just created. We will do so via `_Layout.cshtml` file, which resides inside `Shared` folder in the `Views` folder. We will locate the element that represents the navigation bar. This is `ul` element with `navbar-nav` class. Inside this element, we will add the following item:

```
<li class="nav-item">
  <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="WebAss\
emblyClient">WebAssembly</a>
</li>
```

Finally, we just need to make sure that our application middleware gets access to appropriate files. To do so, need to open `Program.cs` file and add the following line before `app.Run()` call:

```
app.UseBlazorFrameworkFiles();
```

This completes our setup of Blazor SignalR client. Next, we will launch it and see it in action.

Launching Blazor client

If you launch your application and navigate to the **WebAssembly** page, you will see that it's almost identical to the home page of the application. When you type a message and click on the button to send it, the server will return it back to the page, like it's demonstrated by the following screenshot:

Figure 3.2 - Blazor WebAssembly client demonstration

But an interesting thing happens if you open the home page of the application in a separate tab. If you do so, any message you send from either of the clients will appear on both clients, as can be seen from the following screenshot:

Figure 3.3 - Broadcasted message appears on all connected clients

This is because we are broadcasting the message to all clients. Any client that is connected to the hub and has an event handler with the same name as the server expects (which is **ReceiveMessage** in our case) will receive the message.

And this concludes the chapter about setting up in-browser SignalR clients. Let's summarize what we've learned.

Summary

Different JavaScript clients work in a similar way. You would first build an object that represents a SignalR connection. Then, you will add event listeners to the object for the calls from the server.

Then you will add functions or event handlers that will allow you to invoke server-side methods from the client. Then you will start the actual connection.

To set up a JavaScript SignalR client, you need to reference an appropriate JavaScript file on your page. You can either add CDN link to a published version of this file, or you can set this file up via NPM.

Blazor WebAssembly client is a type of .NET client. It relies on the same NuGet package that you would use in any type of .NET client.

Blazor WebAssembly allows you to run compiled .NET code in your browser. You can make some modifications to your stand-alone Blazor WebAssembly application to be able to host it inside an existent ASP.NET Core application.

If you have multiple SignalR clients active and you broadcast a message from one of them to all clients, each connected client will receive the message. As long as a client has an appropriate event handler, it will process the message.

In the next chapter, we will have a look at external SignalR client that you would be able to run from stand-alone applications that don't use browsers.

Test yourself

1. What is the name of the NuGet package that contains all necessary components of SignalR client?
   A. Microsoft.AspNetCore.SignalR.Client
   B. AspNetCore.SignalR.Client
   C. Microsoft.SignalR.Client
   D. System.SignalR.Client
2. When you use `HubConnectionBuilder` on either JavaScript or .NET, will it start the connection?
   A. Yes
   B. Not automatically
   C. Only on JavaScript
   D. Only on .NET
3. What is the method on a hub connection object that you need to call to trigger a method on SignalR hub?
   A. InvokeAsync
   B. SendAsync
   C. `SendAsync` on .NET client and `invoke` on JavaScript client
   D. `InvokeAsync` on .NET client and `send` on JavaScript client

Further reading

Official documentation on SignalR JavaScript client: https://docs.microsoft.com/en-us/aspnet/core/signalr/javascript-client

JavaScript client API references: https://docs.microsoft.com/en-us/javascript/api/?view=signalr-js-latest

Official documentation on SignalR .NET client: https://docs.microsoft.com/en-us/aspnet/core/signalr/dotnet-client

.NET client API references: https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.signalr.client

4 - External SignalR clients

In the previous chapter, we have covered the basic of using in-browser clients to connect to a SignalR hub. Even though these clients are suitable to cover a wide range of SignalR use cases, they aren't sufficient to cover them all. Not all of your clients will be running in the browser. And not all your clients will be a part of the same web application.

In this chapter, we will cover external SignalR clients. All of these can be set up from a stand-alone application of any type. The SignalR client can be set up in some background service running on an IoT device. Or it can be in the back-end of a mobile app.

The chapter consists of the following topics:

- Setting up .NET client
- Setting up Java client
- Setting up a raw WebSocket client

By the end of this chapter, you will have learned how to use officially supported SignalR clients in stand-alone application. But, on top of this, you will have learned how to get a bare WebSocket to communicate with a SignalR server. This knowledge will give you the ability to write your own SignalR client in any language that isn't officially supported.

Prerequisites

This chapter assumes that you already have set up your development environment, as described in **chapter 1**. You will need the following:

- A machine with either Windows, Mac OS or Linux operating system
- A suitable IDE or code editor (Visual Studio, JetBrains Rider or VS Code)
- .NET 6 SDK (or newer)

Also, since we are continuing to build on top of the application that we have developed in the previous chapter, we need the code that we have written previously. If you have skipped the previous chapter, you can access the complete code from the following location in the GitHub repository:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide/tree/main/Chapter-03/Part-02/LearningSignalR

The complete code samples from this chapter are available from the following location in the GitHub repo, which has separate folders corresponding to individual parts of the chapter:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide/tree/main/Chapter-04

Setting up .NET client

In **chapter 3**, we have already set up .NET SignalR client inside of a Blazor WebAssembly application. The process that we will go through now will be similar. For example, we will rely on the same NuGet package. But this time, we will set the client up inside a stand-alone console application.

Setting up .NET console app as a SignalR client

The first thing that we will need to do is to create our console application. Because this would be a stand-alone application without any dependencies shared with the other projects in your solution, it's up to you whether you create it inside the solution folder and add it to the solution. Keeping the new project inside the solution may make it easier to manage, but it isn't strictly necessary.

Once you have selected the folder that you will create the console application in, execute the following command inside that folder to instantiate the project:

```
dotnet new console -o DotnetClient
```

Now, we will need to add a SignalR NuGet package to it. To do it, open your command line terminal inside the project folder and execute the following command:

```
dotnet add package Microsoft.AspNetCore.SignalR.Client
```

Next, we will apply some SignalR client logic to our `Program.cs` file inside of the project. To do so, we will delete all existing content from this file. Then, we will add the following statement to reference the namespace of SignalR client library:

```
using Microsoft.AspNetCore.SignalR.Client;
```

Next, we will add a prompt for the full SignalR hub URL and will create a `hubConnection` object based on it:

```
Console.WriteLine("Please specify the URL of SignalR Hub");

var url = Console.ReadLine();

var hubConnection = new HubConnectionBuilder()
                        .WithUrl(url)
                        .Build();
```

Next, we will map `ReceiveMessage` event to the `hubConnection` object. Every time the server-side hub would trigger this event, the message sent from the server would be written in the console:

```
hubConnection.On<string>("ReceiveMessage",
    message => Console.WriteLine($"SignalR Hub Message: {message}"));
```

After this, we will add a loop that will allow us to carry on using the client until we explicitly type `exit`. Every message that we type will be sent as a parameter to `BroadcastMessage` method on the SignalR hub:

```
try
{
    await hubConnection.StartAsync();

    while (true)
    {
        var message = string.Empty;

        Console.WriteLine("Please specify the action:");
        Console.WriteLine("0 - broadcast to all");
        Console.WriteLine("exit - Exit the program");

        var action = Console.ReadLine();

        Console.WriteLine("Please specify the message:");
        message = Console.ReadLine();

        if (action == "exit")
            break;

        await hubConnection.SendAsync("BroadcastMessage", message);
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
    Console.WriteLine("Press any key to exit...");
    Console.ReadKey();
    return;
}
```

The above code consist of our SignalR logic and basic error handling, which is always a good idea to do to prevent our application from crashing unexpectedly.

Now, we can launch our application and see it in action.

Testing our .NET client

First, we will need to launch our SignalR server application. We can do so by executing `dotnet run` command from `SignalRServer` project folder. Then, once the application is up and running, we can launch the SignalR client console application by executing `dotnet run` command from `DotnetClient` project folder.

When we are prompted to enter the SignalR hub URL, we can enter the URL listed in `applicationUrl` section of `launchSettings.josn` file of `SignalRServer` project followed by `/learningHub`. So, for example, if your base application URL is `https://localhost:7128`, then the URL you need to enter is `https://localhost:7128/learningHub`.

For the testing purposes, we may want to open the home page of our web application to get JavaScript client running in the browser. Because both applications have been set up to broadcast message to all connected SignalR clients, what you will see is that whenever you enter a message in your console application, in-browser application will receive it. Likewise, if you send a message from your in-browser application, your console application will receive it. This can be seen on the following screenshot:

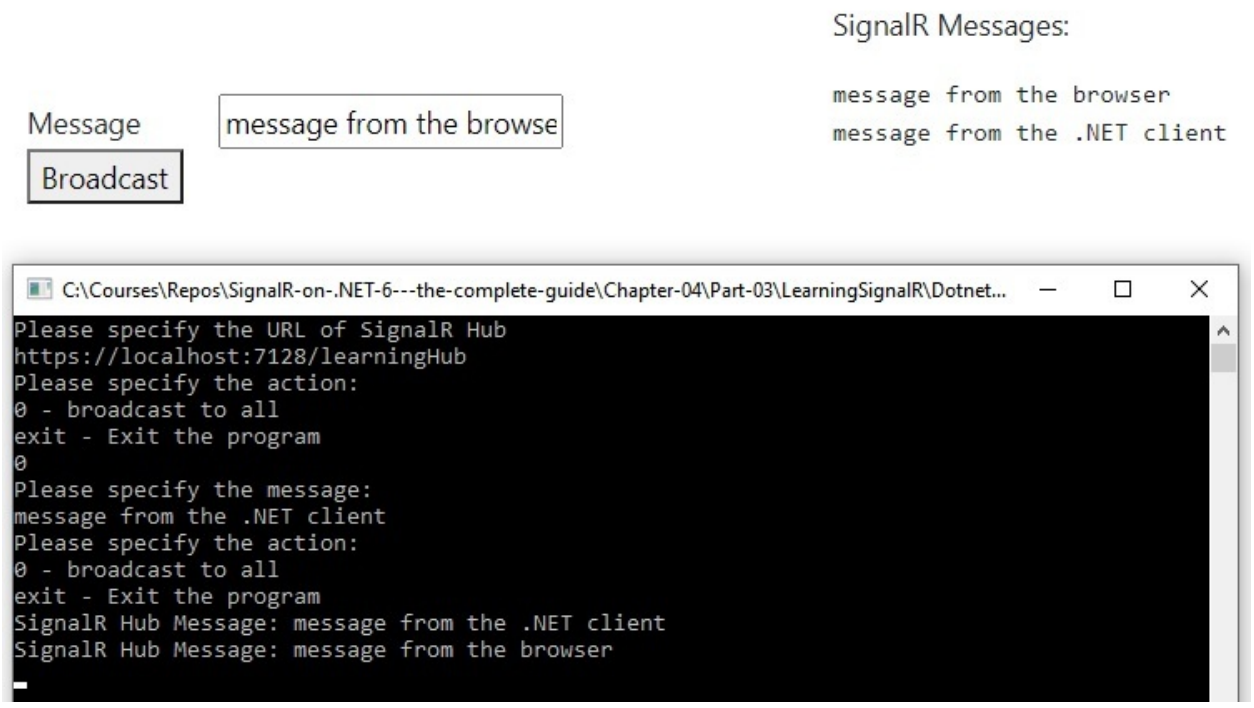

Figure 4.1 - .NET console application can communicate with JavaScript client

And this demonstrates the basics of running SignalR client inside a stand-alone .NET application. Next, we will cover another supported client type - Java client. If you are not a Java developer and you never intend to build a Java client, you can skip this section. But because Java is a widely used and universal language, this information will still be useful.

Setting up Java client

Just like .NET 6, Java is a universal language that can be used on any of the popular operating systems. To use it, you will first need to install Java SDK. There are multiple ways you can do it. But if you already have .NET environment set up, perhaps the easiest way to do it is to download it via VS Code extension pack[13]. Although there are several Java IDEs, VS Code is more than adequate as a code editor.

Once you have installed Java SDK, you will need to install one of its build tools. The most popular of these are Gradle and Maven. They work differently, but the outcome will be roughly the same. You will end up with the same Java code. And you will be able to compile and run your application either way.

We will be using Maven package repository to download our SignalR client library from. But Gradle can access it too. So it doesn't matter which build tool you use. It's outside the scope of this book to provide a detailed instruction on how to install and use either Gradle or Maven. But referenced to detailed user manuals are available in the **further reading** section of this chapter.

Setting up Java project

Once you have your build tool set up, we need to instantiate a Java project. This process will be different depending on whether you are using Gradle or Maven.

Generating a project template with Gradle

If you are using Gradle, you need to run the following command inside any folder of your choice:

```
1 gradle init
```

Then, you will be asked to provide various parameters to your project. You can select anything for most of the options (or just use defaults), but do make sure that the language that you have selected is Java and the project type is `application`. With these options selected, it will instantiate your project with some code already being inside.

The SignalR package will work with other languages, such as Kotlin. After all, it's a JVM language, just like Java. But since Java has been out there for much longer, this is the language that you will see the samples in.

Generating a project template with Maven

With Maven, you need to generate a project from a so-called archetype. It's equivalent to .NET project template. And perhaps one of the simplest archetypes is `maven-archetype-quickstart`. To generate a project from it, you can use the following command:

[13] https://code.visualstudio.com/docs/languages/java

```
mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes -DarchetypeArt\
ifactId=maven-archetype-quickstart -DarchetypeVersion=1.4
```

Whether you have used Gradle or Maven, you should now have a folder structure that contains `App.java` file in it with `main` method. This is equivalent to `Program.cs` in C#. And this is the file we will be adding SignalR client to. But first we will need to add a reference to a relevant dependency.

Adding SignalR client Java dependency

You will first need to visit the following page and verify what is the latest version of SignalR package is:

https://search.maven.org/artifact/com.microsoft.signalr/signalr

Then you will need to click on that version number to open the page, which will provide you dependency insertion syntax for various build manager types. For example, assuming that the latest version of the version package is 6.0.0, this is the markup you will need to insert into the `dependencies` section of the `pom.xml` file if you are using Maven:

```
<dependency>
  <groupId>com.microsoft.signalr</groupId>
  <artifactId>signalr</artifactId>
  <version>6.0.0</version>
</dependency>
```

This would be the entry you would need to insert into `dependencies` section of `gradle.build` file if you have used Groovy DSL during project setup:

```
implementation 'com.microsoft.signalr:signalr:6.0.0'
```

This would be the entry you would need to insert into `dependencies` section of `gradle.build.kts` file if you have used Kotlin DSL during project setup:

```
implementation("com.microsoft.signalr:signalr:6.0.0")
```

And there are quite a few other options, as Maven and Gradle are far from being the only build tools for Java.

Once you have added the dependencies, you may want to remove the default tests (if you have any), as those would cause the build to break after you've changed the content of `App.java` file. And now we are ready to start modifying the code.

Adding SignalR client code to Java application

We will open our `App.java` file and ensure that we have all of the following `import` statements in it:

```
import com.microsoft.signalr.HubConnection;
import com.microsoft.signalr.HubConnectionBuilder;
import java.util.Scanner;
```

We will then need to ensure that the signature of the `main` method looks like follows:

```
public static void main(String[] args) throws Exception {
```

Inside this method, we prompt the user to enter the URL of the SignalR hub:

```
System.out.println("Please specify the URL of SignalR Hub");
Scanner reader = new Scanner(System.in);
String input = reader.nextLine();
```

We can't read textual input from the console directly, like we can in C#. Therefore we are using `Scanner` class for it. Then we build our `hubConection` object and map `aReceiveMessage` event to it:

```
HubConnection hubConnection = HubConnectionBuilder.create(input)
        .build();

hubConnection.on("ReceiveMessage", (message) -> {
    System.out.println(message);
}, String.class);
```

Then we start the connection and getting the application to send any messages that we type to `BroadcastMessage` method on the SignalR hub. At any point, we can type `exit` and this will stop the connection:

```
hubConnection.start().blockingAwait();

while (!input.equals("exit")){
    input = reader.nextLine();
    hubConnection.send("BroadcastMessage", input);
    }

hubConnection.stop();
```

`reader` object (which is an instance of `Scanner`) that we have created earlier just keeps reading the messages from the console. And this completes the setup of our Java application. We can now launch it to see it in action.

Launching Java SignalR client

To launch our `SignalRServer` application, all we have to do is execute `dotnet run` command from its project folder. But with a Java application, building and launching it will depend on the build manager that you use. It could be as simple as executing `gradle run` command. But it could be more complicated if you use any other build tools. So please check the documentation of your build tool for the exact command that you need to execute.

Another caveat is that, depending on configuration, Java HTTP client may not work with development HTTPS certificate. If this is the case, then the easiest way to resolve it is to remove the HTTPS URL from `applicationUrl` entry of `launchSetting.json` file of your `SignalRServer` application.

Once you launch the application, you can verify that it can send and receive messages. The following screenshot shows an example of the application that was set up with Kotlin DSL on Gradle. The launch command was `gradle run -q --console=plain`, which makes it run as a plain console.

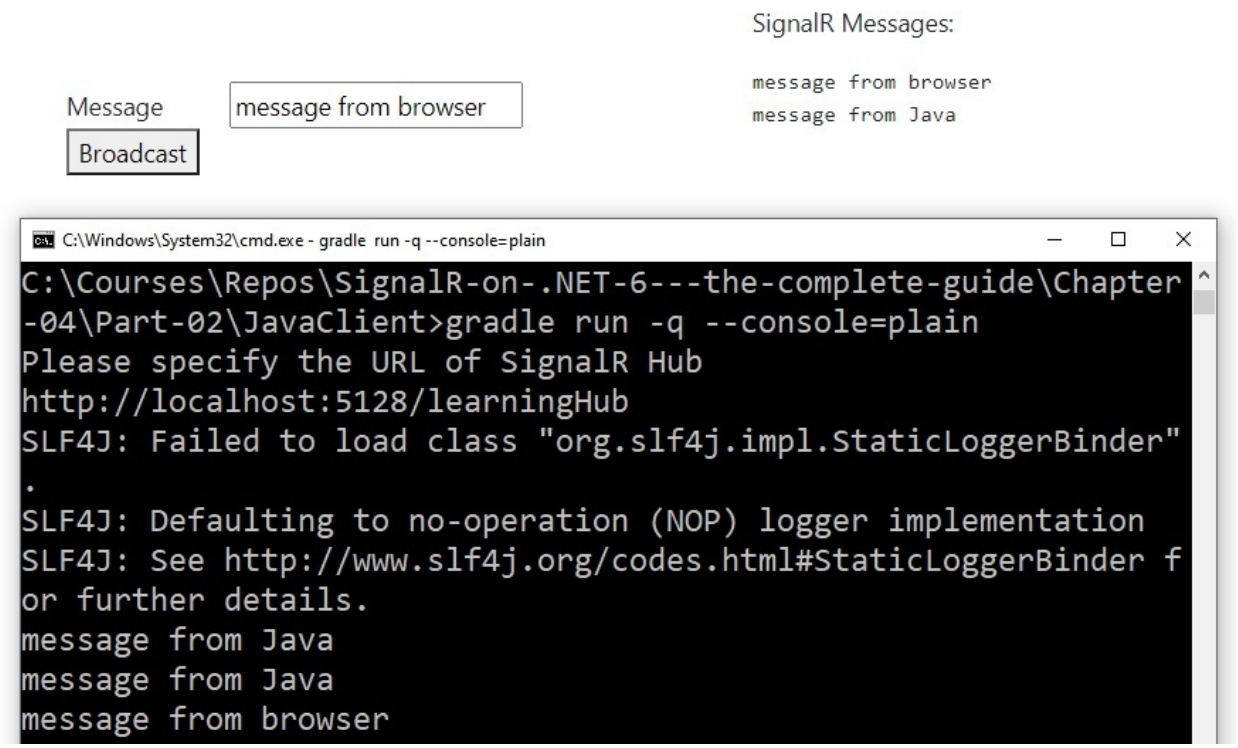

Figure 4.2 - Java client in action

You can safely ignore any errors related to `slf4j`. These errors simply mean that no default logging provider was found. You will need to enable an additional third-party package to make these errors disappear.

And this concludes our overview of Java client, which, at the time of writing, is the only client that is supported outside of .NET. But the good news is that SignalR supports raw WebSockets too, which can be written in any language. And this is what we will have a look at next.

Setting up a raw WebSocket client

We have now covered all SignalR client types that Microsoft officially supports. But what if the technology you want to use is not on the list? Also, what if there is no way to actually write a SignalR client and you need to connect an existing WebSocket client to it?

Well, the good news is that you can connect a raw WebSocket to SignalR hub. And once it is connected, you can easily see the structure of the messages that are being exchanged. This will allow you to write your own SignalR client implementation in any language of your choice, as WebSocket is a standard protocol which you can write code for in any language. But today we will focus on .NET implementation of it.

Setting up WebSocket client

We will create another project and call it `WebSocketClient`. You may choose to include this project in an existing solution. But you don't have to, as it will not share any direct dependencies with any of the projects inside your solution.

The project will be based on .NET **Console Application** template. To initiate the project, execute the following command in a folder of your choice:

```
dotnet new console -o WebSocketClient
```

You won't need to add any external dependencies to the project at all. WebSocket library is already included in .NET `System` library.

The first thing we will need to do in the new project is delete all existing content from the `Program.cs` file. Then, we will add references to WebSocket and text processing namespaces. To do so, add the following `using` statements:

```
using System.Net.WebSockets;
using System.Text;
```

Next, we will add some code to prompt the user to enter WS protocol URL pointing at the SignalR hub endpoint:

```
Console.WriteLine("Please specify the URL of SignalR Hub with WS/WSS protocol");
var url = Console.ReadLine();
```

WebSocket uses WS instead of HTTP in the URLs (and WSS instead of HTTPS). But otherwise, it uses the same TCP/IP protocol as HTTP. Therefore you will be able to point it at the same addresses and ports. So, your SignalR hub endpoint URL will hardly change. Instead of being `https://{base URL}/learningHub`, it will become `wss://{base URL}/learningHub`.

Next, we will connect our WebSocket and send the initial request to SignalR hub. Our request is nothing more than a handshake in JSON format that SignalR hub expects. It defines the protocol and the version.

```
try
{
    var ws = new ClientWebSocket();

    await ws.ConnectAsync(new Uri(url), CancellationToken.None);

    var handshake = new List<byte>(Encoding.UTF8.GetBytes(@"{""protocol"":""json"", \
""version"":1}"))
                {
                    0x1e
                };

    await ws.SendAsync(new ArraySegment<byte>(handshake.ToArray()), WebSocketMessage\
Type.Text, true, CancellationToken.None);

    Console.WriteLine("WebSockets connection established");
    await ReceiveAsync(ws);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
    Console.WriteLine("Press any key to exit...");
    Console.ReadKey();
    return;
}
```

As you can see, we cannot just send human-readable text over WebSocket. We need to convert it to bytes. So, as this example clearly shows, SignalR makes your job much easier compared to raw WebSocket programming.

The last statement that we have entered inside the `try` block is a call to `ReceiveAsync` method. We will add this method now:

```
static async Task ReceiveAsync(ClientWebSocket ws)
{
    var buffer = new byte[4096];

    try
    {
        while (true)
        {
            var result = await ws.ReceiveAsync(new ArraySegment<byte>(buffer), Cance\
llationToken.None);
```

```
            if (result.MessageType == WebSocketMessageType.Close)
            {
                await ws.CloseOutputAsync(WebSocketCloseStatus.NormalClosure, string\
.Empty, CancellationToken.None);
                break;
            }
            else
            {
                Console.WriteLine(Encoding.Default.GetString(Decode(buffer)));
                buffer = new byte[4096];
            }
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
        Console.WriteLine("Press any key to exit...");
        Console.ReadKey();
        return;
    }
}
```

In this method, we are using a byte buffer of a fixed size. We then just carry on listening on WebSocket connection, while populating this buffer with bytes that we receive from it. Every time we receive some data, we reset the buffer and start again. We do this until WebSocket gets closed by the server.

Before we can actually convert bytes to human-readable message, we use `Decode` method. And this is what this method consists of:

```
static byte[] Decode(byte[] packet)
{
    var i = packet.Length - 1;
    while (i >= 0 && packet[i] == 0)
    {
        --i;
    }

    var temp = new byte[i + 1];
    Array.Copy(packet, temp, i + 1);
    return temp;
}
```

This method is necessary, because we use buffer of a fixed size, which will inevitably contain empty

bytes. If we just attempt to convert it into text as is, the empty bytes will be converted too. There is actually a textual symbol that represents them. And this would make our message less readable. To prevent this from happening, we are removing all empty bytes from our message before we process it.

And this concludes the basic setup of WebSocket client. Even though we have only added the most basic WebSocket functionality, the code that we have written is already fairly complicated. This is why you shouldn't use raw WebSocket client unless you absolutely have to. Because SignalR exists, writing raw WebAssembly is almost like writing a web application in Assembler.

Now, we can launch our WebSocket application and see it in action.

Launching WebSocket client

We will now launch our `SignalRServer` project by executing `dotnet run` command in the project directory. Then we will execute the same command inside `WebSocketClient` project directory.

When our WebSocket client application launches, it will ask us to provide the URL to the SignalR endpoint. But remember that we are dealing with WebSocket protocol here and not with HTTP. Therefore, although we can use our original Hub URL, we will need to replace `https` prefix with `wss` (or, if you are using `http` URL, replace it with `ws`).

We don't have any listeners in the WeBSocket client. But we still receive the messages. And, if the connection was successful, what you will notice right away is that we have received an empty object from the server. This was a normal part of the handshake.

But interesting things will start to happen when we start broadcasting messages from the home page of our SignalR Server application. Our WebSocket client will pick them up, but it won't be just the content of the message itself. It will come as a JSON with some other fields, like it can be seen from the following screenshot:

Figure 4.3 - WebSocket client picks up detailed messages from the server

In this JSON object, we have `type` field, which is set to 1. Then we have `target` field, which is set to the name of the event handler. And then we have `arguments` field, containing a collection of parameters. And this is how SignalR middleware knows what events to trigger and what parameters to apply. It's a simple format. And if you get familiar with it, you will be able to write your own SignalR client implementation in any language of your choice.

But another interesting thing happens when you leave the application running for a while. You will start occasionally receiving a JSON message that has nothing in it except `type` field with the value of 6. Well, this is a heartbeat message. And its purpose is to ensure that the connection is still operational. `type` field tells the system what kind of message it is. It's 1 for a standard message and 6 for heartbeat. There are some more values, but you will need to study the code[14] of the library to learn them. After all, it's open source and publicly available.

And this completes our overview of SignalR clients. Let's summarize what we have learned.

Summary

In this chapter, you have learned how to set up a SignalR client inside a stand-alone .NET application. To do so, you will need to install a NuGet package with the SignalR client library. It's the same process for all .NET client types, including Blazor WebAssembly or a mobile app.

We have covered how to set up a Java client. To do so, you will need to install SignalR client package from Maven central package repository. Once done, you will be able to set up SignalR hub connection in any type of Java application back-end.

Finally, we have covered the use of raw WebSocket as a SignalR client. Even though we only had a look at .NET example, the overall principles of WebSocket programming will be the same in any

[14] https://github.com/SignalR/SignalR

language. And, as we so, SignalR messages are nothing more than nicely formatted JSON, we can use WebSocket programming to write a client library for it in any language that isn't officially supported.

In the next chapter, we will start adding some complexity to our SignalR clients. You will learn how to group clients together and how to send messages from the server-side hub to specific clients and not just broadcast them to all.

Test yourself

1. Which SignalR client types are officially supported?
   A. JavaScript
   B. .NET
   C. Java
   D. All of the above
2. What data can you read from SignalR messages if you connect a raw WebSocket to it?
   A. You cannot, as all data in encrypted
   B. Only the name of event handlers
   C. Message type, the name of the event handlers and full message payload
   D. Detailed message metadata, including the headers
3. If you write a SignalR client in Java, which type of build tool can you use?
   A. Gradle
   B. Maven
   C. Neither of the above
   D. Both of the above

Further reading

Official documentation of SignalR Java client: https://docs.microsoft.com/en-us/aspnet/core/signalr/java-client

Java client API references: https://docs.microsoft.com/en-us/java/api/com.microsoft.signalr

Gradle user manual: https://docs.gradle.org/current/userguide/userguide.html

Maven documentation: https://maven.apache.org/guides/

.NET WebSocket programming: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/websockets

5 - Sending messages to individual clients or groups of clients

So far, we have only covered one type of SignalR call from the server-side hub to the clients - broadcasting a message to all clients. But this is far from being the only way you can use SignalR. The hub allows you to send messages to individual clients. You can also group clients together and send messages to specific groups of clients.

But even if you broadcast messages to all clients, you don't necessary have to include the client that the message has originated from. After all, the client already knows what the message is. So you can exclude it from the list of recipients.

The chapter consists of the following topics:

- Broadcasting messages to all clients
- Sending messages to specific clients
- Working with client groups

By the end of this chapter, you will have learned how to be selective on which clients you want to send messages to from SignalR hub.

Prerequisites

This chapter assumes that you already have set up your development environment, as described in **chapter 1**. You will need the following:

- A machine with either Windows, Mac OS or Linux operating system
- A suitable IDE or code editor (Visual Studio, JetBrains Rider or VS Code)
- .NET 6 SDK (or newer)

Also, since we are continuing to build on top of the application that we have developed in the previous chapter, we need the code that we have written previously. If you have skipped the previous chapter, you can access the complete code from the following location in the GitHub repository:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide/tree/main/Chapter-04/Part-03/LearningSignalR

The complete code samples from this chapter are available from the following location in the GitHub repo, which has separate folders corresponding to individual parts of the chapter:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide/tree/main/Chapter-05

Broadcasting messages to all clients

In the previous chapters, we have already broadcasted SignalR messages to all connected clients. This was achieved by making the call on `Clients.All` property in the server-side hub. But this way of broadcasting a message has its limitations. What if you want to exclude the client that has sent the message from the list of its recipients?

In SignalR, this can be achieved by using `Clients.Others` instead of `Clients.All`. And this is what we will now implement.

Open `LearningHub.cs` file in your `SignalRServer` project and add the following method to the class:

```
public async Task SendToOthers(string message)
{
    await Clients.Others.ReceiveMessage(message);
}
```

So, as you can see, this method is identical to `BroadcastMessage` we had previously, except for one little detail. And now we need to get our clients to call it.

Applying changes to JavaScript client

We will modify our JavaScript client first. And we will start by adding new controls to the markup of the page. let's open `Index.cshtml` file, which is located in `Home` folder inside the `Views` folder. We will locate `div` element with the value of `class` attribute set to `col-md-4` and insert the following markup into the element:

```
<div class="control-group">
    <div>
        <label for="others-message">Message</label>
        <input type="text" id="others-message" name="others-message" />
    </div>
    <button id="btn-others-message">Send to Others</button>
</div>
```

We will now need to add JavaScript that the newly added button will trigger to make the relevant call. To do so, we will open `site.js` file, which is located inside `js` folder of `wwwroot` folder. We will add the following `click` event hanler to the file:

```
$('#btn-others-message').click(function () {
    var message = $('#others-message').val();
    connection.invoke("SendToOthers", message).catch(err => console.error(err.toStri\
ng()));
});
```

Our JavaScript client is now ready. What we need to do next is apply changes to our .NET client too. We will not be modifying all clients, as both `BlazorClient` and `DotnetClient` projects are based on same technologies and use the same libraries. So, we will choose `DotnetClient` project to modify, as it is more different from the JavaScript client that we have already updated.

Updating .NET Client

In `DotnetClient` project, open `Program.cs` file and replace the content of the `try` block with the following:

```
await hubConnection.StartAsync();

var running = true;

while (running)
{
    var message = string.Empty;

    Console.WriteLine("Please specify the action:");
    Console.WriteLine("0 - broadcast to all");
    Console.WriteLine("1 - send to others");
    Console.WriteLine("exit - Exit the program");

    var action = Console.ReadLine();

    Console.WriteLine("Please specify the message:");
    message = Console.ReadLine();

    switch (action)
    {
        case "0":
            await hubConnection.SendAsync("BroadcastMessage", message);
            break;
        case "1":
            await hubConnection.SendAsync("SendToOthers", message);
            break;
```

```
27          case "exit":
28              running = false;
29              break;
30          default:
31              Console.WriteLine("Invalid action specified");
32              break;
33      }
34  }
```

So, we now have two actions - our original call to `BroadcastMessage` hub method and a call to `SendToOthers` method. You can select the appropriate action by typing either `0` or `1` in the console.

Now, both of our clients are ready. Let's launch our application and see it in action.

Testing exclusive broadcasting functionality

To now see how sending messages to others works, we will need to launch both SignalR clients. One of the easiest ways of doing so is to execute `dotnet run` command from both `SignalRServer` and `DotnetClient` folder.

Once both of your applications are up and running, you will need to manually connect `DotnetClient` application to the SignalR hub of `SignalRServer`. To do so, when prompted by the console, type in the application address, which can be found under `applicationUrl` entry of `launchSettings.json` file of `SignalRServer` project. Then, append `learningHub` path to the URL. So, if your application URL is `https://localhost:7128`, the address the you need to specify is `https://localhost:7128/learningHub`.

Now, once you open the home page of the web application in your browser and your console app is connected to the SignalR hub, you can test how different SignalR endpoints work. If you select an action that triggers `SendToOthers` method on the hub, your original client won't receive the message, but your other client will. If, however, you trigger `BroadcastMessage` method, both of your clients will receive the message. This can be clearly seen on the following screenshot:

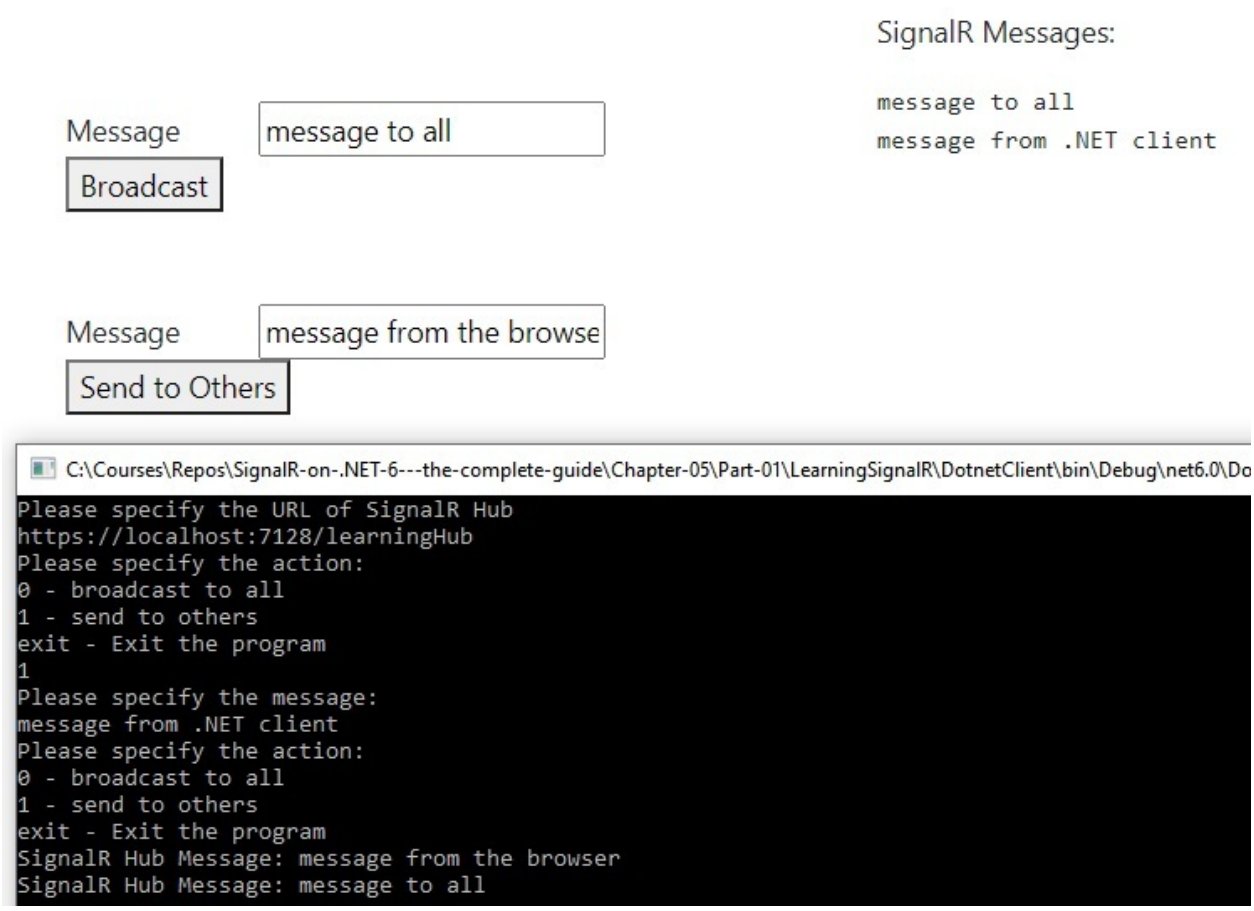

Figure 5.1 - Messages to others aren't being received by the sender

Next, you will learn how to send messages to individual clients and not just broadcast them to all.

Sending messages to specific clients

In the previous section, you have learned how to send SignalR messages to all connected clients except the sender. Now, we will do the opposite. We will send messages only back to the sender. There is in-built property on `Clients` property of the `Hub` base class that allows us to do this. It's called `Caller`.

There are many different reasons why you would want to send message only to the client that has triggered a particular method. For example, any scenario where request-response messaging model is appropriate would utilize this functionality. Let's now apply this ability, so we can see it in action.

Enabling self-messages

To utilize the `Caller` property, we will add the following method to the `LearningHub` class of `SignalRServer` project:

```
public async Task SendToCaller(string message)
{
    await Clients.Caller.ReceiveMessage(GetMessageToSend(message));
}
```

Then, we will add the following markup to the `div` element with `col-md-4` class inside `Index.cshtml` file in `Home` folder inside the `Views` folder:

```
<div class="control-group">
    <div>
        <label for="self-message">Message</label>
        <input type="text" id="self-message" name="self-message" />
    </div>
    <button id="btn-self-message">Send to Self</button>
</div>
```

Then, we will add the following `click` event handler to the `site.js` file:

```
$('#btn-self-message').click(function () {
    var message = $('#self-message').val();
    connection.invoke("SendToCaller", message).catch(err => console.error(err.toStri\
ng()));
});
```

And finally, we will replace the content of the `while` statement inside `Program.cs` file of `DotnetClient` with the following:

```
var message = string.Empty;

Console.WriteLine("Please specify the action:");
Console.WriteLine("0 - broadcast to all");
Console.WriteLine("1 - send to others");
Console.WriteLine("2 - send to self");
Console.WriteLine("exit - Exit the program");

var action = Console.ReadLine();

Console.WriteLine("Please specify the message:");
message = Console.ReadLine();

switch (action)
{
```

```
16      case "0":
17          await hubConnection.SendAsync("BroadcastMessage", message);
18          break;
19      case "1":
20          await hubConnection.SendAsync("SendToOthers", message);
21          break;
22      case "2":
23          await hubConnection.SendAsync("SendToCaller", message);
24          break;
25      case "exit":
26          running = false;
27          break;
28      default:
29          Console.WriteLine("Invalid action specified");
30          break;
31  }
```

Now, if we launch the applications and try to utilize `SendToCaller` method from either of the clients, we can see that it's only the client itself that receives the message. The other client doesn't receive anything, as can be seen on the following screenshot:

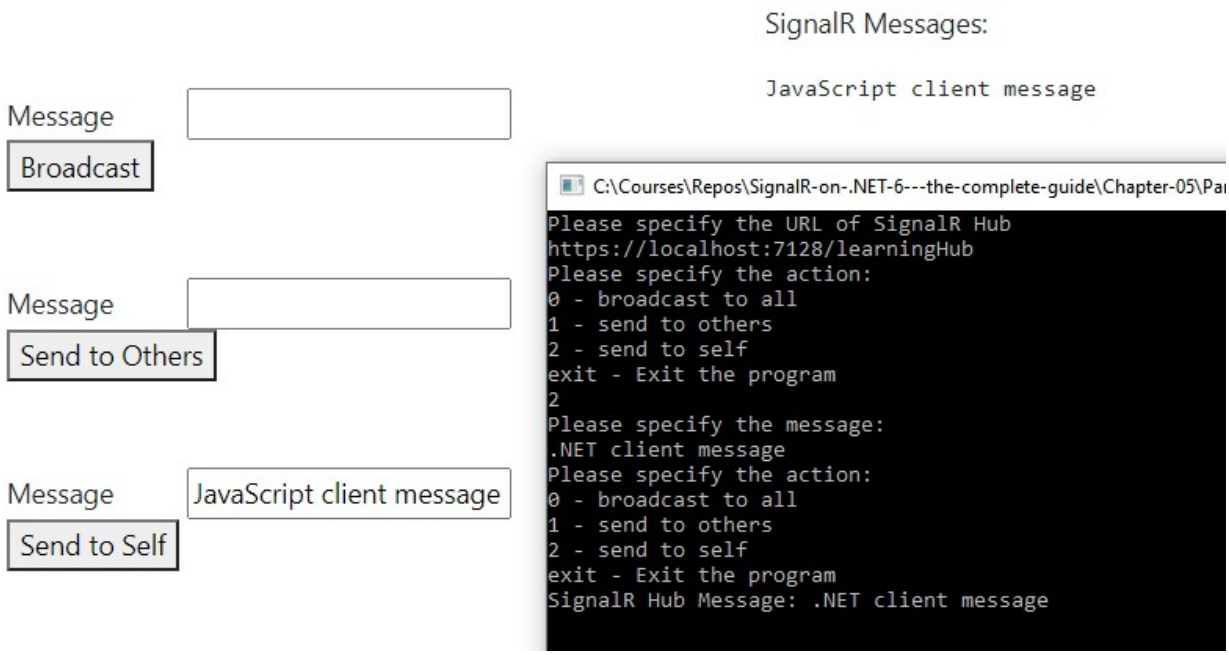

Figure 5.2 - Clients sending messages to themselves

But sending message back to itself is not the only way to send messages to individual clients in SignalR hub. You can actually identify a specific client (or multiple clients) and send messages to

those. And this is what we will have a look at next.

Sending messages to other clients

SignalR Hub has a property calle `Context`. This property represent the context of the current connection and contains some metadata related to it. For example, if you are connecting as an authenticated user (which we will talk about in **chapter 8**), you will be able to get user information from this property.

One of the properties of `Context` is called `ConnectionId`. And this is the property that contains an auto-generated string that represents a unique identifier of the current client connection. If you know a unique identifier of any specific client connection, then you will be able to send messages to specific clients. And this is what we will do next.

Modifying SignalR hub

We will start by adding the following method to `LearningHub` class of `SignalRServer` project:

```
private string GetMessageToSend(string originalMessage)
{
    return $"User connection id: {Context.ConnectionId}. Message: {originalMessage}";
}
```

`Context.ConnectionId` will reveal the connection identifier of the current client, so other clients will be able to send messages to it. Then, we will utilize this method in all of our client calls by replacing all instances of `ReceiveMessage(message)` with the following:

```
ReceiveMessage(GetMessageToSend(message))
```

Then we will add the following method to the hub:

```
public async Task SendToIndividual(string connectionId, string message)
{
    await Clients.Client(connectionId).ReceiveMessage(GetMessageToSend(message));
}
```

In this method, we are selecting a specific client by passing a connection id to `Client` method of `Clients` property. But we could also select more than one client to send the message to. There is also `Clients` method on the `Clients` property. And this method allows you to use either a single connection id string or a collection of them. For example, if we had multiple connection ids that we wanted to send a message to, we could store them in `connectionIds` variable that represents a C# collection (for example, `List<string>`). With this, we could implement the following call:

```
await Clients.Clients(connectionIds).ReceiveMessage(GetMessageToSend(message));
```

Now, we will add necessary components to both of our clients.

Modifying the clients

In the markup of `Index.cshtml` class, we will insert the following markup next to the other control groups:

```
<div class="control-group">
    <div>
        <label for="individual-message">Message</label>
        <input type="text" id="individual-message" name="individual-message" />
    </div>
    <div>
        <label for="connection-for-message">User connection id:</label>
        <input type="text" id="connection-for-message" name="connection-for-message"\
 />
    </div>
    <button id="btn-individual-message">Send to Specific User</button>
</div>
```

Then, we will insert the following `click` event handler to the `site.js` file:

```
$('#btn-individual-message').click(function () {
    var message = $('#individual-message').val();
    var connectionId = $('#connection-for-message').val();
    connection.invoke("SendToIndividual", connectionId, message).catch(err => consol\
e.error(err.toString()));
});
```

Finally, we will replace the content of the `while` statement inside `Program.cs` file of `DotnetClient` with the following:

```
var message = string.Empty;

Console.WriteLine("Please specify the action:");
Console.WriteLine("0 - broadcast to all");
Console.WriteLine("1 - send to others");
Console.WriteLine("2 - send to self");
Console.WriteLine("3 - send to individual");
Console.WriteLine("exit - Exit the program");

var action = Console.ReadLine();

Console.WriteLine("Please specify the message:");
message = Console.ReadLine();

switch (action)
{
    case "0":
        await hubConnection.SendAsync("BroadcastMessage", message);
        break;
    case "1":
        await hubConnection.SendAsync("SendToOthers", message);
        break;
    case "2":
        await hubConnection.SendAsync("SendToCaller", message);
        break;
    case "3":
        Console.WriteLine("Please specify the connection id:");
        var connectionId = Console.ReadLine();
        await hubConnection.SendAsync("SendToIndividual", connectionId, message);
        break;
    case "exit":
        running = false;
        break;
    default:
        Console.WriteLine("Invalid action specified");
        break;
}
```

Now, let's launch our applications to see how they work.

Seeing individual client messages in action

After we launch our applications and connecting the `DotnetClient` app to the SignalR hub, we can first sent initial message from either of the clients to obtain its connection id. Then, we can send a message from the other client to it by specifying this connection id. This is demonstrated by the following screenshot:

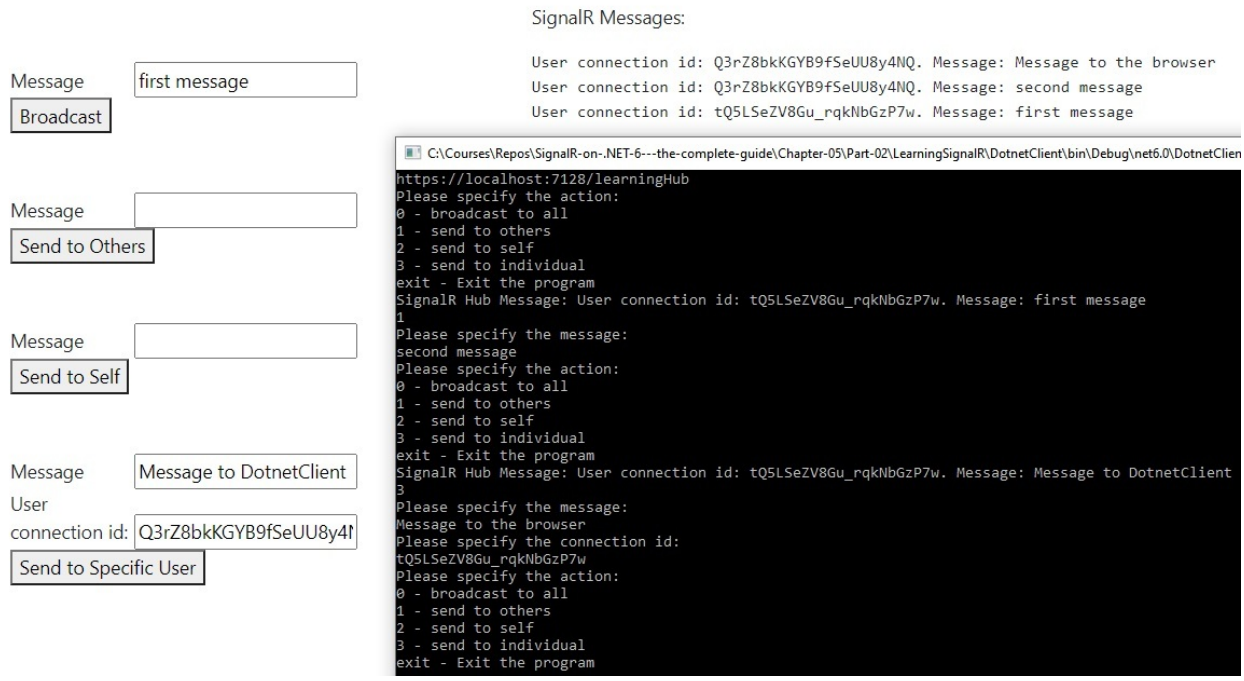

Figure 5.3 - Clients sending messages to a specific client

But by now, you have probably noticed a problem with this approach. Even though connection id reliably identifies an individual client, it's not very convenient to use from the user's perspective. Plus, if the client disconnects and then reconnects, the connection id would have changed, so you will no longer be able to send messages to the client until you know the updated connection id value.

But there is a solution for that. You can map a connection id with some nice human-readable name. You can use your own dictionary in the back-end of the server, which would be the best solution under certain circumstances. But in many cases, you can just use inbuilt mechanism for this, which is SignalR client groups.

Working with client groups

SignalR Hub has `Groups` property, which allows you to add clients to a group and remove clients from it. Each group is one-to-many relationship between an arbitrary group name and a collection

of connection ids. `Clients` property of the `Hub` base class has `Group` method, which allows you to specify a group name and send a message to all clients in the group.

Groups are useful in many scenarios. You can use them to assign all clients of a specific category to a group. Or you can just associate a group with a specific user. Then, you won't lose the track of the connected client that represents the user. The user name you will specify will always map to the right connection id, as long as the client is connected. Plus, an individual user may have multiple clients connected simultaneously. For example, you may be using the same social network platform in multiple tabs of your browser, while simultaneously having it open in a mobile app.

So, let's modify our hub to see how groups can be used.

Using SignalR groups inside the hub

We will start by adding the following methods to the `LearningHub` class of `SignalRServer` project:

```
public async Task SendToGroup(string groupName, string message)
{
    await Clients.Group(groupName).ReceiveMessage(GetMessageToSend(message));
}

public async Task AddUserToGroup(string groupName)
{
    await Groups.AddToGroupAsync(Context.ConnectionId, groupName);
    await Clients.Caller.ReceiveMessage($"Current user added to {groupName} group");
    await Clients.Others.ReceiveMessage($"User {Context.ConnectionId} added to {grou\
pName} group");
}

public async Task RemoveUserFromGroup(string groupName)
{
    await Groups.RemoveFromGroupAsync(Context.ConnectionId, groupName);
    await Clients.Caller.ReceiveMessage($"Current user removed from {groupName} grou\
p");
    await Clients.Others.ReceiveMessage($"User {Context.ConnectionId} removed from {\
groupName} group");
}
```

In the code above, when we are adding a current client to a specific named group (or removing it from a specific group), we are sending messages about this action to all clients. But the message sent to the calling client will be different from what is sent to other clients.

Now, we will modify our `OnConnectedAsync` and `OnDisconnectedAsync` event handlers. We will add every one of our clients to `HubUsers` group when it connects and remove it from this group when it disconnects. This is purely to demonstrate how clients can be assigned to groups automatically.

Our hub is now ready. Let's now modify our clients to be able to use groups.

Enabling SignalR clients to use groups

First, we will add the following control groups to the appropriate section of `Index.cshtml` class:

```
<div class="control-group">
    <div>
        <label for="group-message">Message</label>
        <input type="text" id="group-message" name="group-message" />
    </div>
    <div>
        <label for="group-for-message">Group Name</label>
        <input type="text" id="group-for-message" name="group-for-message" />
    </div>
    <button id="btn-group-message">Send to Group</button>
</div>
<div class="control-group">
    <div>
        <label for="group-to-add">Group Name</label>
        <input type="text" id="group-to-add" name="group-to-add" />
    </div>
    <button id="btn-group-add">Add User to Group</button>
</div>
<div class="control-group">
    <div>
        <label for="group-to-remove">Group Name</label>
        <input type="text" id="group-to-remove" name="group-to-remove" />
    </div>
    <button id="btn-group-remove">Remove User from Group</button>
</div>
```

Then, we will add the following `click` handlers to `site.js` file:

```
$('#btn-group-message').click(function () {
    var message = $('#group-message').val();
    var group = $('#group-for-message').val();
    connection.invoke("SendToGroup", group, message).catch(err => console.error(err.\
toString()));
});

$('#btn-group-add').click(function () {
    var group = $('#group-to-add').val();
    connection.invoke("AddUserToGroup", group).catch(err => console.error(err.toStri\
ng()));
});

$('#btn-group-remove').click(function () {
    var group = $('#group-to-remove').val();
    connection.invoke("RemoveUserFromGroup", group).catch(err => console.error(err.t\
oString()));
});
```

Finally, we will update our `Program.cs` file of the `DotnetClient` project. We will replace the content of the `while` statement with the following:

```
var message = string.Empty;
var groupName = string.Empty;

Console.WriteLine("Please specify the action:");
Console.WriteLine("0 - broadcast to all");
Console.WriteLine("1 - send to others");
Console.WriteLine("2 - send to self");
Console.WriteLine("3 - send to individual");
Console.WriteLine("4 - send to a group");
Console.WriteLine("5 - add user to a group");
Console.WriteLine("6 - remove user from a group");
Console.WriteLine("exit - Exit the program");

var action = Console.ReadLine();

if (action != "5" && action != "6")
{
    Console.WriteLine("Please specify the message:");
    message = Console.ReadLine();
}

```

```
22  if (action == "4" || action == "5" || action == "6")
23  {
24      Console.WriteLine("Please specify the group name:");
25      groupName = Console.ReadLine();
26  }
27
28  switch (action)
29  {
30      case "0":
31          await hubConnection.SendAsync("BroadcastMessage", message);
32          break;
33      case "1":
34          await hubConnection.SendAsync("SendToOthers", message);
35          break;
36      case "2":
37          await hubConnection.SendAsync("SendToCaller", message);
38          break;
39      case "3":
40          Console.WriteLine("Please specify the connection id:");
41          var connectionId = Console.ReadLine();
42          await hubConnection.SendAsync("SendToIndividual", connectionId, message);
43          break;
44      case "4":
45          hubConnection.SendAsync("SendToGroup", groupName, message).Wait();
46          break;
47      case "5":
48          hubConnection.SendAsync("AddUserToGroup", groupName).Wait();
49          break;
50      case "6":
51          hubConnection.SendAsync("RemoveUserFromGroup", groupName).Wait();
52          break;
53      case "exit":
54          running = false;
55          break;
56      default:
57          Console.WriteLine("Invalid action specified");
58          break;
59  }
```

And this is it. Our code is ready. Now, if we launch both `SignalRServer` and `DotnetClient` applications, then you will be able to play with the groups. As this screenshot demonstrates, we can add clients to groups, remove them from groups and send messages to specific groups:

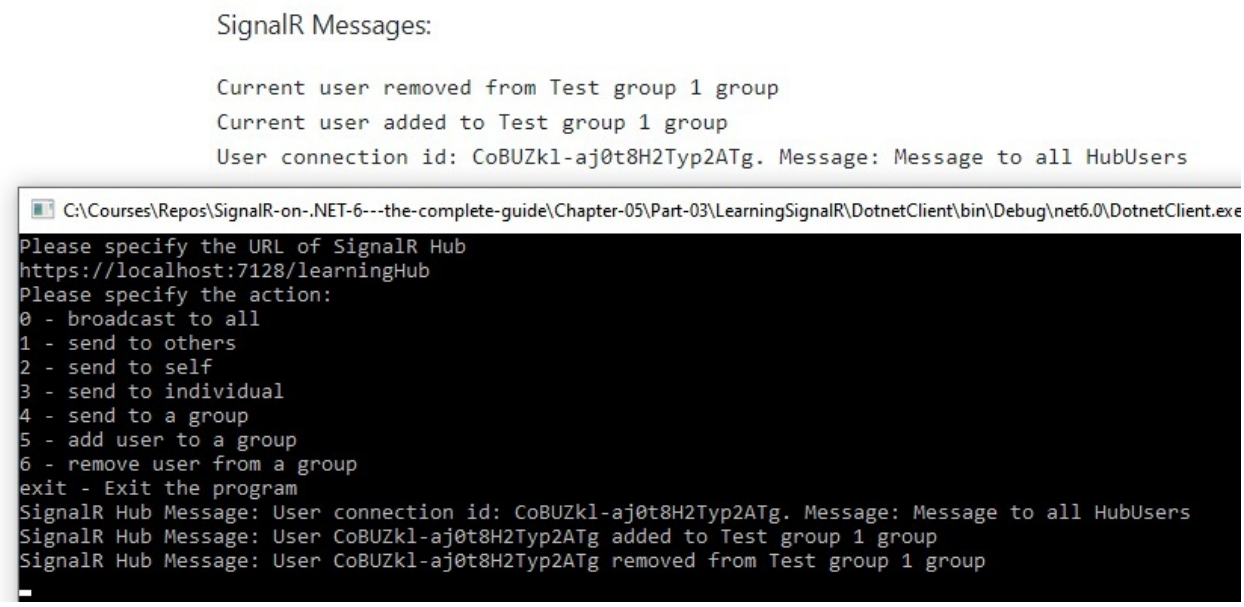

Figure 5.4 - Using SignalR groups

And this concludes the chapter about sending SignalR messages to specific clients. Let's now summarize what we have learned.

Summary

In this chapter, you have learned that, when you choose to broadcast a message to all clients, there is a convenient setting in the SignalR hub that allows you to exclude the sender of the message from the list of the recipients.

You have also learned that there are multiple ways of sending messages from SignalR hub to individual users. You can send message back to the sender. Or you can send message to specific clients by specifying its connection id. Or you can specify multiple connection ids and send the message to more than one client simultaneously.

But the easiest way to send messages to multiple clients at a time is to explicitly add clients to groups. Unlike connection ids, group names can be easily readable. And you can easily add clients to or remove clients from groups at will.

So far, we have only covered how to send individual messages in every call. In the next chapter, you will learn how to use streaming in SignalR, which is the way to send multiple messages through an open channel.

Test yourself

1. What property of the Hub base class represents a unique identifier of an individual client?

   A. ConnectionId
   B. Clients.ConectionId
   C. Context.ConnectionId
   D. ClientId
2. How can you map a client connection identifier in SignalR hub to a human-readable name?
   A. Use a custom lookup dictionary
   B. Use SignalR group
   C. All of the above
   D. This is not possible
3. When will unique client connection identifier change for a given client
   A. Never
   B. When the client disconnects and reconnects
   C. Every time the client makes a call
   D. When the client is added to a group

Further reading

Manage users and groups in SignalR: https://docs.microsoft.com/en-us/aspnet/core/signalr/groups

Coordinating IoT cluster with SignalR: https://scientificprogrammer.net/2020/10/30/coordinating-iot-cluster-with-signalr/

6 - Streaming in SignalR

Until this point, we have only covered the use of singular messages in SignalR. But SignalR also supports message streaming.

Streaming is when, instead of sending all data at ones, you send it in chunks. Perhaps, it's a large object that gets split into smaller parts. Or it could be a collection of object, where each item is sent individually. There are many scenarios where you can use streaming.

The chapter consists of the following topics:

- What is streaming used for
- Client streaming in SignalR
- Server streaming in SignalR

By the end of this chapter, you will have learned how to use both client-to-server and server-to-client streaming in SignalR.

Prerequisites

This chapter assumes that you already have set up your development environment, as described in **chapter 1**. You will need the following:

- A machine with either Windows, Mac OS or Linux operating system
- A suitable IDE or code editor (Visual Studio, JetBrains Rider or VS Code)
- .NET 6 SDK (or newer)

Also, since we are continuing to build on top of the application that we have developed in the previous chapter, we need the code that we have written previously. If you have skipped the previous chapter, you can access the complete code from the following location in the GitHub repository:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide/tree/main/Chapter-05/Part-03/LearningSignalR

The complete code samples from this chapter are available from the following location in the GitHub repo, which has separate folders corresponding to individual parts of the chapter:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide/tree/main/Chapter-06

What is streaming used for

So, why would you even need to use streaming? A more specific question is why would you use streaming in SignalR? After all, it's not gRPC, where streaming is perhaps the only way to deal with live-processed collections. In SignalR, you can just send an individual item of such a collection as soon as it becomes available. You already have a persistent connection open. Well, there are many real-life scenarios where streaming would still be useful, even in SignalR.

Let's first have a look at one of the best-known use of streaming, which is the transfer of video data. When you watch a video in your browser, your browser doesn't wait to download it. If it had to wait for the download to finish, you would have spent a long time waiting before you were able to watch a movie. High-quality movies are gigabytes in size. So, even with a fast internet connection, it would still take a noticeable amount of time to download.

So, instead of downloading the entire video all at once, the app that you use to watch the video on downloads it in small chunks. And, as a viewer, you won't even notice, as long as the download is happening quicker than the playback. You won't really care about the part of the video an hour from now. You just need enough data from the video to cover the next minute or so. So the experience of watching the video is completely seamless.

There is also a scenario where there is never such a thing as a complete video. For example, with close-circuit television (CCTV) cameras, the video feed is continuous. So it is with any type of live streaming. The recipients of the data just carry on receiving chunks of data while the camera is filming. In this scenario, streaming is literally the only thing you can do.

Why raw data cannot be sent as individual messages

SignalR streaming functionality allows you to implement features like live streaming. It's not uncommon to include the ability to do live video calls in chat applications that SignalR was designed to support. Likewise, it's not uncommon to connect IoT devices to CCTV. A SignalR client can be used on these types of devices to then transfer this data to the server.

So why can't you just send the chunks of data as individual messages? Well, this is because doing so would be extremely inefficient. If you recall from **chapter 4**, every SignalR message has a JSON envelope. So, if there are many messages that you would need to send within a short period of time (as would be inevitable with live video streaming), there would be excessive amount of data to deal with. You would have to use additional network bandwidth to transfer all this data. Plus, there would be unnecessary computational overhead separating the data from the rest of the JSON envelope. If you are dealing with many of such messages, both the network bandwidth overhead and the computational overhead can accumulate to such an extent that your application will no longer be usable.

Plus, don't forget that you are dealing with raw bytes. Those collections of bytes will need to be re-assembled into something that the user can consume. They fit each other in a continuous manner.

And it will be fairly difficult to figure out which byte arrays are supposed to connect together if they are all dealt by independent event handlers that are intended to handle singular calls.

Streaming greatly simplifies this process. The data is dealt with in the same order as it arrives, so it's easy to reassemble this data on the other end. And, as long as the streaming channel is open, you won't have to clog it with any additional metadata.

But what about dealing with collections of complete object? After all, each object can already be read and there's already a persistent SignalR connection open. So why not just send each one of them individually? Well, it appears that, under certain scenarios, it still makes sense to stream such collections.

Why streaming collections also makes sense in SignalR

Yes, some scenarios under which it would be appropriate to use streaming in gRPC would not warrant for the use of streaming in SignalR. In gRPC, streaming is, for example, the only way to allow the server to send messages to the client without being prompted. But SignalR does it by design.

But there are still scenarios in SignalR where you would want objects to be streamed rather than sent individually. The same principles apply as before. If the objects come in a quick succession and there are many of them, then streaming will save you the overhead of using JSON envelope for every individual message. You will use less network bandwidth and you would be able to deal with the messages much quicker. And, as before, if you use streaming, it would be easier to process the messages in the right order.

So, to summarize, the key benefits of streaming in SignalR are the following:

- Improved performance due to lack of JSON envelope in every message
- Easy to maintain message order

SignalR can stream from the client to the server and from the server to the client. We will have a look at both, starting with client-to-server streaming.

Client streaming in SignalR

In this type of streaming, we open a stream from the client. The client will then carry on sending messages via the stream until it runs out the messages or the stream is closed. The server will receive the messages in the same order as they arrive and within the same call context.

Under normal circumstances, i.e., if there are no failures, it's the client that controls the stream. Server can forcibly close it if needed, but it's the client that initiates a normal closure of the stream.

To make it easier to demonstrate client streaming in practice, it's the clients that we will modify first.

Applying client streaming functionality to JavaScript client

We will not add any new HTML markup to the index page of the `SignalRServer` web application. Instead, we will modify the existing `click` handler in JavaScript.

In `site.js` file that is located inside `js` folder of `wwwroot` directory, locate the `click` handler for `#btn-broadcast` selector and replace the code with the following:

```
$('#btn-broadcast').click(function () {
    var message = $('#broadcast').val();

    if (message.includes(';')) {
        var messages = message.split(';');

        var subject = new signalR.Subject();
        connection.send("BroadcastStream", subject).catch(err => console.error(err.t\
oString()));
        for (var i = 0; i < messages.length; i++) {
            subject.next(messages[i]);
        }

        subject.complete();

    } else {
        connection.invoke("BroadcastMessage", message).catch(err => console.error(er\
r.toString()));
    }
});
```

So, this is what we are doing here. If the text that we enter doesn't contain any semicolon (;) characters, we execute a singular call to `BroadcastMessage` method, as we did before. Otherwise, we split the text into an array, trigger `BroadcastStream` method on the hub and open a stream to it. We stream all the items from the array until there are no more items left in it. And then we close the stream.

Client-side stream is created by instantiating a `Subject` type, which is a part of `signalR` object from the SignalR JavaScript library. Then, we call `send` method on the object that represents SignalR connection. The parameters of this call are the name of the method on the hub (which we will add shortly) and the `subject` object that we have created earlier. This object, while it's active, represents an abstraction of the open stream.

Next, we are placing items on the stream by passing them as parameters into `next` function of `subject` object. The server should receive each of these items instantly, not taking into account any network latency. Then, once there are no more items left to send, we close the stream by calling `complete` function of the `subject`.

Next, we will apply client streaming functionality to our .NET client.

Applying client streaming functionality to .NET client

We will open `Program.cs` file of our `DotnetClient` project and the first thing we will add to it is the following namespace reference:

```
using System.Threading.Channels;
```

Then, we need to find the `switch` statement and replace the `case "0"` statement with the following:

```
case "0":
    if (message?.Contains(';') ?? false)
    {
        var channel = Channel.CreateBounded<string>(10);
        await hubConnection.SendAsync("BroadcastStream", channel.Reader);

        foreach (var item in message.Split(';'))
        {
            await channel.Writer.WriteAsync(item);
        }

        channel.Writer.Complete();
    }
    else
    {
        hubConnection.SendAsync("BroadcastMessage", message).Wait();
    }
    break;
```

We are doing a similar thing to what we have been doing in the JavaScript client. We are opening a channel, which is represented by `Channel` class from `System.Threading.Channels` namespace. Then we make a call to `BroadcastStream` method on the hub and pass the reader of the channel to it. This represents a stream it can read from. Then, we write all items of the collection into the stream. And finally, once we are done with it, we close the stream.

When we are calling `CreateBounded` method on the `Channel` class, we are specifying `string` as the data type. This is to tell the channel that we will be placing `string` messages in it. Otherwise, it could be any data type.

This was done to demonstrate how different client types implement client streaming in SignalR. Now, we need to add an appropriate method to our SignalR hub.

Adding client streaming listener to SignalR hub

In our `LearningHub.cs` file of `SignalRServer` project, we will add the following method:

```
public async Task BroadcastStream(IAsyncEnumerable<string> stream)
{
    await foreach (var item in stream)
    {
        await Clients.Caller.ReceiveMessage($"Server received {item}");
    }
}
```

The method simply accepts `IAsyncEnumerable` object as a parameter. The type of the data is `string` because we are receiving `string` messages. But the data type could be anything.

`IAsyncEnumerable` collection allows us to place items into the collection while we iterate through its existing items. It's similar to reading from a stream. In fact, this is exactly what it represents in this particular case.

Because the iterator of `IAsyncEnumerable` expects an item to be added to the collection at any time, it awaits for an item to be added instead of just exiting the loop when there are no more items. And this is what `await foreach` statement represents.

All we do in our method is read all items from the client stream and, for each of these, send the message back to the caller, telling it that we have processed it. Let's now build and launch our applications to see how client streaming works.

Testing client streaming functionality

First, we need to launch `SignalRServer` application. You can either do it from an IDE, or you can execute `dotnet run` command from inside the project folder. When the application is up and running, you can navigate to its home page in the browser, enter some message into the first input box, insert some semicolons into the message and see the output that you receive from the hub. It should look similar to what's displayed on the following screenshot:

Figure 6.1 - Results of client streaming in JavaScript client

Then, if we launch `DotnetClient` project, connect the console to the hub endpoint at `{base URL}/learningHub`, and choose option "0" when prompted, we will be able to enter any arbitrary text with semicolons and view the results of the client stream that was processed by the server. It should look similar to the output shown by the following screenshot:

```
Please specify the URL of SignalR Hub
https://localhost:7128/learningHub
Please specify the action:
0 - broadcast to all
1 - send to others
2 - send to self
3 - send to individual
4 - send to a group
5 - add user to a group
6 - remove user from a group
exit - Exit the program
0
Please specify the message:
first message;second message;third message;fourth message
Please specify the action:
0 - broadcast to all
1 - send to others
2 - send to self
3 - send to individual
4 - send to a group
5 - add user to a group
6 - remove user from a group
exit - Exit the program
SignalR Hub Message: Server received first message
SignalR Hub Message: Server received second message
SignalR Hub Message: Server received third message
SignalR Hub Message: Server received fourth message
```

Figure 6.2 - Results of client streaming in .NET client

This concludes our overview of client streaming functionality. Next, we will have a look at server streaming.

Server streaming in SignalR

Server streaming, as the name suggests, works the other way. You still need a client involved. The client needs to trigger a server streaming endpoint on the server. And once it's triggered, it will then subscribe to the stream and carry on reading messages from it either until there are no more messages left or the server disconnects the stream.

We will now add some server streaming functionality to our clients and the server-side hub. This time, we will start with the server, as it's the server that controls the stream.

Adding server streaming capabilities to SignalR hub

In `LearningHub.cs` file inside `SignalRServer` project, we will add the following namespace reference:

```
using System.Runtime.CompilerServices;
```

Then, we will add the following method:

```
public async IAsyncEnumerable<string> TriggerStream(
    int jobsCount,
    [EnumeratorCancellation]
    CancellationToken cancellationToken)
{
    for (var i = 0; i < jobsCount; i++)
    {
        cancellationToken.ThrowIfCancellationRequested();
        yield return $"Job {i} executed successfully";
        await Task.Delay(1000, cancellationToken);
    }
}
```

In this method, we first receive an integer `jobCounts` parameter. This is just a normal parameter, like the ones used by any other type of SignalR hub methods. It can be anything.

The next parameter is cancellation token. It has `EnumeratorCancellation` attribute on it. I won't delve too deep into details of what it does. But in this case, it allows the method to trigger cancellation if the calling client has issued a cancellation. You can think of it like passing a cancellation token to a method, but doing so remotely over the network.

Then, we look at the `jobCount` parameter and make as many iterations as this parameter specifies. In each iteration, we write a message into the stream by using `yield return` statement. And we wait one second between the iterations. The stream will close as soon as we have iterated through all messages.

So, our server is done. Let's make necessary changes to the clients.

Adding server streaming listener to JavaScript client

We will first add some HTML markup to our `Index.cshtml` file, which resides inside `Home` folder of `Views` folder. We will add the following control group alongside other elements with `control-group` class:

```
<div class="control-group">
    <div>
        <label for="number-of-jobs">Number of Jobs</label>
        <input type="text" id="number-of-jobs" name="number-of-jobs" />
    </div>
    <button id="btn-trigger-stream">Trigger Server Stream</button>
</div>
```

We will then add the following event handler to `site.js` file in `js` folder of `wwwroot`:

```
$('#btn-trigger-stream').click(function () {
    var numberOfJobs = parseInt($('#number-of-jobs').val(), 10);

    connection.stream("TriggerStream", numberOfJobs)
        .subscribe({
            next: (message) => $('#signalr-message-panel')
                        .prepend($('<div />').text(message))
        });
});
```

So, here is what we are doing. We are calling `stream` method on `connection` object and we are passing the name of the SignalR hub method we want to execute (`TriggerStream`) and `numberOfJobs` parameter. Then, we subscribe to the stream and keep outputting the messages that we received from it, until there are no more.

So, our JavaScript client is now done. Let's move on to the .NET client and make all necessary changes to it.

Adding server streaming listener to .NET client

We will now open `Program.cs` file of `DotnetClient` project. We will locate the block of code that puts `6 - remove user from a group` on the screen. Then, we insert the following statement in the line below:

```
Console.WriteLine("7 - trigger a server stream");
```

Next, we will insert the following `case` statement into the `switch` expression:

```
case "7":
    Console.WriteLine("Please specify the number of jobs to execute.");
    var numberOfJobs = int.Parse(Console.ReadLine() ?? "0");
    var cancellationTokenSource = new CancellationTokenSource();
    var stream = hubConnection.StreamAsync<string>(
        "TriggerStream", numberOfJobs, cancellationTokenSource.Token);

    await foreach (var reply in stream)
    {
        Console.WriteLine(reply);
    }
    break;
```

So, essentially, we are doing the same thing we've done in JavaScript client, but in a different syntax. We are calling `StreamAsync` method on `hubConnection` object. In this method, we are specifying the data type of `string`. This represent the data type of each item we expect to receive from the stream. Then, once we have created a 'stream' variable, we just iterate through all of the items in the stream until there are no more. And we write every message we receive into the console.

So, now both of our clients are done. Let's launch the applications and see how they behave.

Testing server streaming functionality

We will first launch our `SignalRServer` application. Once it's loaded, we will navigate to its homepage and use the new control group we have created. We will enter an arbitrary number in `Number of Jobs` field and click on the button next to it.

We should now see our screen being populated with massages with one second interval between them. This is what you should expect your page to look like:

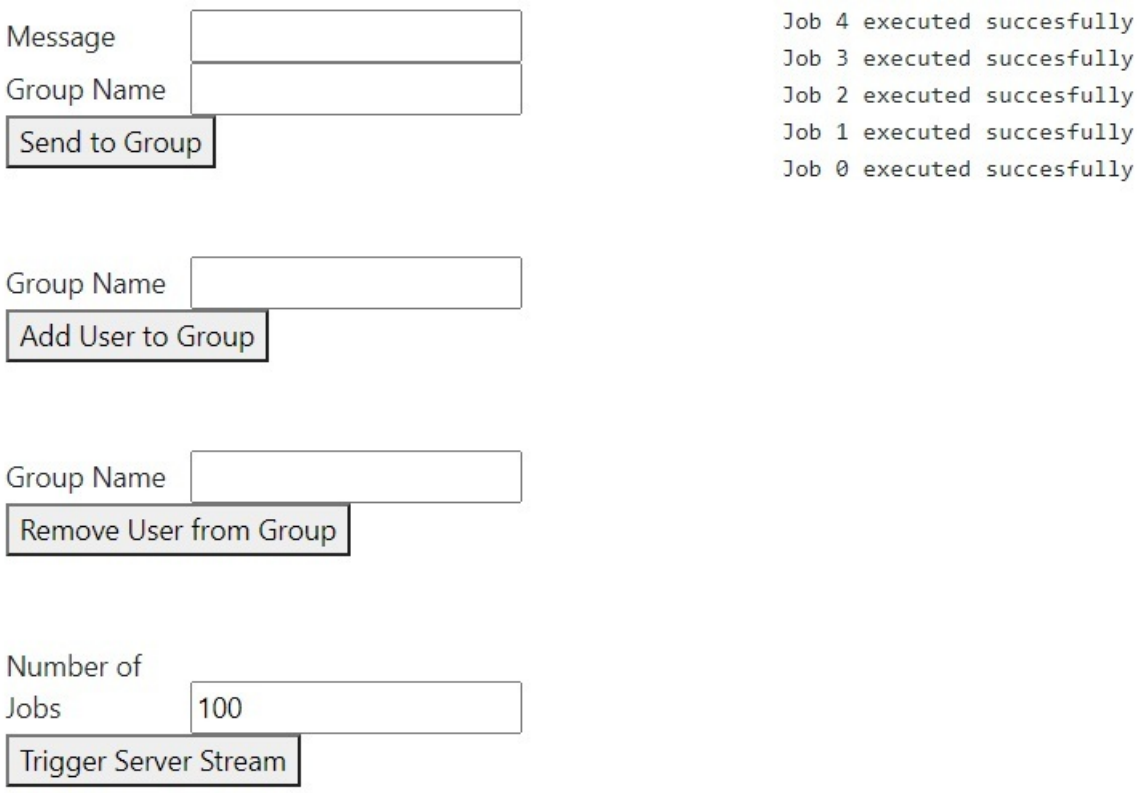

Figure 6.3 - Results of server streaming in JavaScript client

Then, we will launch `DotnetClient` application. Once it's up and running, we will connect it to the fully qualified URL of the SignalR hub and select action `7` from the list. After specifying an arbitrary number of jobs to execute, we should be able to see messages appearing in our console with one second interval between them. It should look like this:

```
Please specify the URL of SignalR Hub
https://localhost:7128/learningHub
Please specify the action:
0 - broadcast to all
1 - send to others
2 - send to self
3 - send to individual
4 - send to a group
5 - add user to a group
6 - remove user from a group
7 - trigger a server stream
exit - Exit the program
7
Please specify the number of jobs to execute.
10
Job 0 executed succesfully
Job 1 executed succesfully
Job 2 executed succesfully
Job 3 executed succesfully
Job 4 executed succesfully
Job 5 executed succesfully
Job 6 executed succesfully
Job 7 executed succesfully
Job 8 executed succesfully
Job 9 executed succesfully
```

Figure 6.4 - Results of server streaming in .NET client

And this concludes our overview of streaming functionality in SignalR. Let's summarize what we have learned in this chapter.

Summary

In this chapter, you have learned the importance of streaming in SignalR. You have learned that streaming messages is much more efficient than sending individual messages in terms of computation, code complexity and network bandwidth usage.

You have learned that SignalR supports both client-to-server streaming and server-to-client. In client-to-server streaming, client fully controls the stream. Unless there is a failure, it's the client that opens the stream, writes data into it and closes the stream.

In server-to-client streaming, client still needs to initiate the streaming process. Client calls a relevant endpoint and subscribes to the streaming channel. And it keeps listening until there are no more messages being sent.

In the next chapter, you will learn how to configure SignalR both from the server and the client perspective. We will cover how to make the communication process more efficient, how to explicitly select transport mechanism and how to perform other types of fine-tunning on the middleware.

Test yourself

1. What is streaming used for?
   A. Transferring a large object in small chunks
   B. Transferring a collection of object by individual items
   C. All of the above
   D. None of the above
2. How would you trigger client streaming in SignalR?
   A. Send client stream to a relevant hub endpoint
   B. Trigger a relevant hub endpoint and subscribe to it
   C. Send a HTTP request to trigger HubContext
   D. None of the above
3. How would you trigger server streaming in SignalR?
   A. Send client stream to a relevant hub endpoint
   B. Trigger a relevant hub endpoint and subscribe to it
   C. Send a HTTP request to trigger HubContext
   D. All of the above

Further reading

Official SignalR streaming documentation: https://docs.microsoft.com/en-us/aspnet/core/signalr/streaming

Low-level documentation on TCP socket streaming: http://etutorials.org/Programming/Pocket+pc+network+programming/Chapter+1.+Winsock/Streaming+TCP+Sockets/

Additional high-level information on streaming: https://www.cloudflare.com/en-gb/learning/video/what-is-streaming/

7 - Advanced SignalR configuration

We have already covered all the fundamental ways of using SignalR messages. We have learned how to set up clients of all supported types, how to send messages to specific clients and how to use data streams, both client-to-server and server-to-client. Now, we will be moving on to more advanced SignalR concepts. And in this chapter, we will cover SignalR configuration.

Most of the time, you won't have to configure SignalR. The default settings are more than sufficient to cover the majority of scenarios where SignalR is used. But sometimes you will have to fine-tune configuration. Some of the settings would allow you to significantly improve the performance or make your application more secure.

SignalR allows you to configure both the server and the client. And both of those have multiple levels of configuration. In this chapter, we will cover them all.

The chapter consists of the following topics:

- Configuring SignalR server
- Configuring SignalR client
- Pros and cons of MessagePack protocol

By the end of this chapter, you will have learned how to configure both SignalR server and the client.

Prerequisites

This chapter assumes that you already have set up your development environment, as described in **chapter 1**. You will need the following:

- A machine with either Windows, Mac OS or Linux operating system
- A suitable IDE or code editor (Visual Studio, JetBrains Rider or VS Code)
- .NET 6 SDK (or newer)
- NPM is installed on the machine, as described in the **chapter 4**.

Also, since we are continuing to build on top of the application that we have developed in the previous chapter, we need the code that we have written previously. If you have skipped the previous chapter, you can access the complete code from the following location in the GitHub repository:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide/tree/main/Chapter-06/Part-03/LearningSignalR

The complete code samples from this chapter are available from the following location in the GitHub repo, which has separate folders corresponding to individual parts of the chapter:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide/tree/main/Chapter-07

To fully understand the JSON settings, you would need to be familiar with JSON format and how you can serialize and deserialize JSON data.

Configuring SignalR server

We will begin by applying some advanced configuration to SignalR server. And before we start, we will need to make sure that `Program.cs` file in `SignalRServer` project has the following namespace references:

```
using Microsoft.AspNetCore.Http.Connections;
using System.Text.Json.Serialization;
```

On the server-side, there are three levels of configuration: top-level SignalR configuration, message protocol configuration and HTTP transport configuration. Let's cover all these categories one by one.

Top-level SignalR configuration

We will begin by locating the following line in `Program.cs` file:

```
builder.Services.AddSignalR();
```

We will then insert the following block into the brackets at the end of `AddSignalR` method call:

```
hubOptions => {
    hubOptions.KeepAliveInterval = TimeSpan.FromSeconds(10);
    hubOptions.MaximumReceiveMessageSize = 65_536;
    hubOptions.HandshakeTimeout = TimeSpan.FromSeconds(15);
    hubOptions.MaximumParallelInvocationsPerClient = 2;
    hubOptions.EnableDetailedErrors = true;
    hubOptions.StreamBufferCapacity = 15;

    if (hubOptions?.SupportedProtocols is not null)
    {
        foreach (var protocol in hubOptions.SupportedProtocols)
            Console.WriteLine($"SignalR supports {protocol} protocol.");
    }
}
```

This code demonstrates all possible configuration options that are available at the context of top-level SignalR configuration. Let's examine them one by one.

KeepAliveInterval

When a client connects to the server, the connection is maintained by sending ping requests to connected clients at regular intervals. This setting controls the interval between the ping requests.

By default, it is set to 15 seconds. Making this interval shorter would allow you to detect a disconnection quicker. Making it longer will save the bandwidth. So, you would decrease this interval when performance and resilience are important and you would increase it when you have to work with limited bandwidth.

MaximumReceiveMessageSize

This setting determines the maximum size of a message that a client can send to the hub. It's represented in bytes and, by default, it's 32 kilobytes.

This setting can be increased if you need to deal with large messages. But if your system is intended to deal with a large quantity of small messages, then it makes sense to decrease this number to increase the performance.

HandshakeTimeout

When client establishes a connection with a SignalR hub, it initiates a handshake. This setting determines how long the server should wait for a response from the client before it gives up on it and considers the connection to be broken. By default, the timeout is 15 seconds.

If you intend to only ever use your system on a fast network, then it may make sense to decrease this number. If the client goes offline during the handshake, your application will detect it quicker. However, if some of your clients are expected to be on slower networks, then it may make sense to increase the interval.

MaximumParallelInvocationsPerClient

By default, each client is only allowed to execute one hub method at a time. The next invocation is queued and is triggered when the current invocation has completed. This setting allows you to change it. It determines the number of invocations a client can do in parallel.

The higher the number - the higher the performance of the clients. But this is achieved at the expense of increased use of computational resources by the server. So it may not be a good idea to use a high number in this field if you have a large quantity of clients and where each client invokes hub methods in a quick succession.

EnableDetailedErrors

This setting determines if the clients would receive internal error information when an exception is thrown during an invocation of a hub method. This setting is disabled by default. We can see how this setting works by replacing `SendToCaller` method of `LearningHub` class with the following:

```
public Task SendToCaller(string message)
{
    throw new Exception("You cannot send messages to yourself.");
}
```

So, if a client invokes this method, an unhandled exception would be thrown. In this case, if we had `EnableDetailedErrors` set to `true`, we would receive the inner exception message by the client that invoked this method. This is demonstrated by the following screenshot, which shows the message being displayed in in-browser console of the JavaScript client:

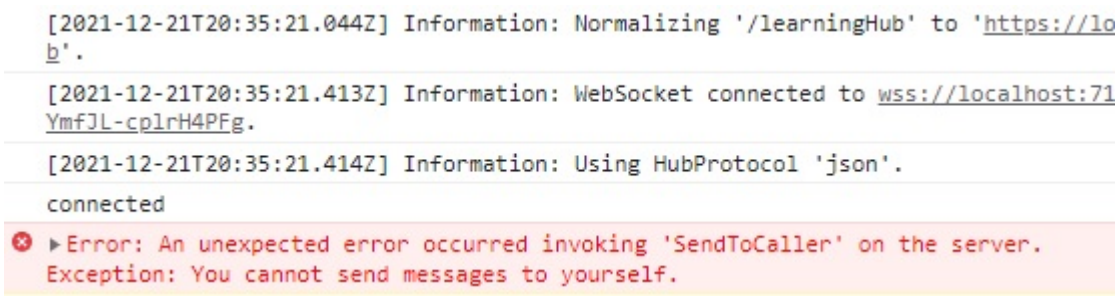

Figure 7.1 - Error message on the client while detailed errors are enabled

If, however, we set `EnableDetailedErrors` set to `false`, the client would still be notified of an error, but the error information would be completely lost, as can be seen in the following screenshot:

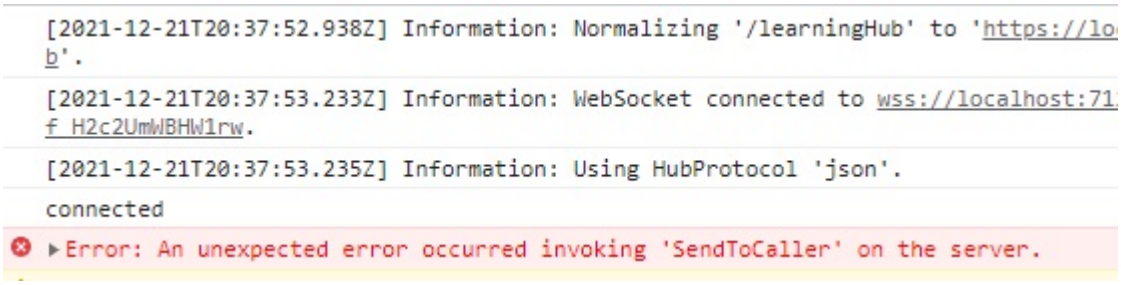

Figure 7.2 - Error message on the client while detailed errors are disabled

This setting should only be used in development environment. It should never be used in production or any other environment that mimics production (integration test, staging, etc). Instead, the actual error should be logged on the server. This is needed to prevent any potential information of inner working of the system from being leaked to malicious users.

StreamBufferCapacity

This setting determines the maximum number of items that can be uploaded into a client-to-server stream. If the limit is reached, the call will be blocked until the existing items are processed.

By default, this number is 10. The higher the number - the faster the server can process the client streams. But this is done at the expense of increased usage of computational resources.

SupportedProtocols

This is a collection of message protocol names that the server supports. The names of these protocols are written as lowercase strings, e.g. `json` and `messagepack`. JSON is supported by default, while MessagePack protocol needs to be explicitly enabled.

This setting is redundant, as there are more intuitive ways of configuring which messaging protocols should be supported. And now we will move on to applying advanced settings for JSON protocol.

JSON message protocol settings

At the end of `AddSignalR` method call, you can replace the semicolon with the following block of code:

```
.AddJsonProtocol(options => {
    options.PayloadSerializerOptions.PropertyNamingPolicy = null;
    options.PayloadSerializerOptions.Encoder = null;
    options.PayloadSerializerOptions.IncludeFields = false;
    options.PayloadSerializerOptions.IgnoreReadOnlyFields = false;
    options.PayloadSerializerOptions.IgnoreReadOnlyProperties = false;
    options.PayloadSerializerOptions.MaxDepth = 0;
    options.PayloadSerializerOptions.NumberHandling = JsonNumberHandling.Strict;
    options.PayloadSerializerOptions.DictionaryKeyPolicy = null;
    options.PayloadSerializerOptions.DefaultIgnoreCondition = JsonIgnoreCondition.Ne\
ver;
    options.PayloadSerializerOptions.PropertyNameCaseInsensitive = false;
    options.PayloadSerializerOptions.DefaultBufferSize = 32_768;
    options.PayloadSerializerOptions.ReadCommentHandling = System.Text.Json.JsonComm\
entHandling.Skip;
    options.PayloadSerializerOptions.ReferenceHandler = null;
    options.PayloadSerializerOptions.UnknownTypeHandling = JsonUnknownTypeHandling.J\
sonElement;
    options.PayloadSerializerOptions.WriteIndented = true;

    Console
    .WriteLine($"Number of default JSON converters: {options.PayloadSerializerOption\
s.Converters.Count}");
});
```

Even though JSON protocol is enabled by default, using `AddJsonProtocol` method allows you to fine-tune it. Pretty much all of the settings are related to payload serialization. And here is what each of these settings does.

PropertyNamingPolicy

This setting determines the policy of naming JSON properties. It used `JsonNamingPolicy` abstract class from `System.Text.Json` namespace. If you need to, you can create your own implementation of it by inheriting from this class. And then you can apply your own implementation to this setting.

Encoder

This property accepts any implementation of `JavaScriptEncoder` class from `System.Text.Encodings.Web` namespace. If your clients use any custom encoding in JSON data that they exchange with the server, you can override this class and apply the custom encoding rules in it.

IncludeFields

This option determines whether or not fields are handled during serialization and deserialization. Normally, just C# properties of your data classes are handled. With this option set to `true`, fields will be handled too.

IgnoreReadOnlyFields

If this option is set to `true` and `IncludeFields` option is also set to `true`, then the fields that are marked as `readonly` will be ignored during serialization. By default, this option is set to `false`.

IgnoreReadOnlyProperties

This option determines whether data class properties that only have a getter and no setter are ignored during serialization. The default value is `false`, so these properties are included.

MaxDepth

This property determines the maximum depth of nested JSON objects used during deserialization. By default, the system will accept 64 levels. If you set the value to `0`, there will be no limit.

NumberHandling

This setting is represented by `JsonNumberHandling` enum from `System.Text.Json.Serialization` namespace. It determines how numeric data in JSON messages is handled. Here are the possible values:

- `Strict` - numbers are only accepted in a standard JSON format with no quotes
- `WriteAsString` - numbers will be written as string surrounded by quotes
- `AllowReadingFromString` - numbers would be readable either from standard JSON numbers or from string values
- `AllowNamedFloatingPointLiterals` - `NaN`, `Infinity`, and `-Infinity` can be read as floating-point constants and translated to their corresponding `double` or `float` numeric values

Normally, you would just use `Strict` option. But there might be some use cases where other options are appropriate.

DictionaryKeyPolicy

This option determines how keys in C# dictionaries get translated to JSON. It's represented by `JsonNamingPolicy` class of `System.Text.Json` namespace, which can be inherited from and overridden.

DefaultIgnoreCondition

This setting controls whether or not properties with default values are ignored during serialization and deserialization. It's represented by `JsonIgnoreCondition` enum of `System.Text.Json.Serialization` namespace, which has the following values:

- `Never` - the properties with default values will always be serialized and deserialized
- `Always` - the properties with default values will always be ignored
- `WhenWritingNull` - a reference-type property or a field will be ignored during serialization when `null` is its value
- `WhenWritingDefault` - a property that has the default value of `null` will be ignored during serialization when `null` is its value

The default value is `Never`.

PropertyNameCaseInsensitive

If this option is set, then the comparison between JSON fields and C# properties won't be case sensitive during deserialization. The default value is `false`.

DefaultBufferSize

This setting determines the default size of a temporary data buffers in bytes. The bigger the buffer - the more data can be transferred in one chunk. But smaller buffers would be useful in an environment with a limited network bandwidth.

ReadCommentHandling

This setting determines how comments are handled in JSON. It's represented by `JsonCommentHandling` enum from `System.Text.Json` namespace and here are the possible values:

`Allow` - the comments are read from JSON
`Skip` - the comment are allowed, but they are not read
`Disallow` - a JSON with comments is considered to be invalid

ReferenceHandler

This setting determines how JSON references are handled. It's represented by `ReferenceHandler` class of `System.Text.Json.Serialization` namespace. The class can be overridden and custom logic can be applied to it.

UnknownTypeHandling

This setting determines how unknown types are handled during deserialization. It's represented by `JsonUnknownTypeHandling` enum from `System.Text.Json.Serialization` namespace and the values are as follows:

- `JsonElement` - a type declared as `object` is deserialized as JSON element
- `JsonNode` - a type declared as `object` is deserialized as JSON node

WriteIndented

If this option is set to `true`, then JSON will be written in a human-readable format with indentations applied. It will be written as a single-line string otherwise. The default value is `false`.

Converters

This setting holds a collection of `JsonSonverter` objects from `System.Text.Json.Serialization` namespace. This overridable class allows you to determine custom conversion rules to transform your JSON data before it's processed. By default, there aren't any converters applied, so the collection is empty.

And this concludes the overview of JSON serialization properties. We will now move on to HTTP transport settings.

Applying advanced transport configuration

We will now locate the line that contains `app.MapHub` and replace it with the following code:

```
app.MapHub<LearningHub>("/learningHub", options =>
{
    options.Transports =
                HttpTransportType.WebSockets |
                HttpTransportType.LongPolling;
    options.CloseOnAuthenticationExpiration = true;
    options.ApplicationMaxBufferSize = 65_536;
    options.TransportMaxBufferSize = 65_536;
    options.MinimumProtocolVersion = 0;
```

```
    options.TransportSendTimeout = TimeSpan.FromSeconds(10);
    options.WebSockets.CloseTimeout = TimeSpan.FromSeconds(3);
    options.LongPolling.PollTimeout = TimeSpan.FromSeconds(10);

    Console
    .WriteLine($"Authorization data items: {options.AuthorizationData.Count}");
});
```

These are the options where you apply transport settings. Let's now examine these settings one by one.

Transports

This setting allows you to set the availability of SignalR transport mechanisms. By default, it supports WebSocket, long polling and server-sent events. In our example above, we've set it to only support WebSocket and long polling.

The values are added together by using bitwise operations.

CloseOnAuthenticationExpiration

This setting determines whether SignalR connection gets closed when authentication expires. If set to `true`, the clients would need to re-authenticate. Otherwise, the connection remains live until the client disconnects.

ApplicationMaxBufferSize

This option represents the maximum buffer size for the data exchange in the application layer in bytes. By default, it's set to 65 kilobytes.

As with any other buffer size settings, the higher the value - the quicker you can transfer the data. But you will be putting a bigger load on your network, so it makes sense to lower this value for low-bandwidth networks.

TransportMaxBufferSize

This option represents the maximum buffer size for the data exchange in the transport layer in bytes. By default, it's set to 65 kilobytes.

MinimumProtocolVersion

This setting determines the minimum protocol version supported by the server. If it's set to 0, then any version would be supported.

TransportSendTimeout

This setting determines how long the application will wait for send action to complete. If this time is exceeded, the connection will be closed.

WebSockets

This is a set of options specific to WebSocket transport mechanism.

LongPolling

This is a set of options specific to long polling transport mechanism.

AuthorizationData

The collection of authorization data used in HTTP pipeline.

And this concludes the overview of SignalR server configuration. Next, we will have a look at how you can configure the clients.

Configuring SignalR client

Different SignalR client types have different settings. But they share some common settings. Also, in all of the client types, you can set some configuration only once when you create a connection object, while other options can be changed at any point.

We will start with an overview of configuration options of a JavaScript client.

Configuring JavaScript client

We will open our `site.js` file, which is located under `js` folder of `wwwroot` folder of `SignalRServer` project. We will find the statement where `connection` object is instantiated and replace it with the following.

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/learningHub", {
        transport: signalR.HttpTransportType.WebSockets | signalR.HttpTransportType.\
LongPolling,
        headers: { "Key": "value" },
        accessTokenFactory: null,
        logMessageContent: true,
        skipNegotiation: false,
        withCredentials: true,
        timeout: 100000
    })
    .configureLogging(signalR.LogLevel.Information)
    .build();
```

Here, we have added a number of configuration options. Let's go over each one of them.

transport

This option determines the transport mechanisms that the client supports. As on the server, you can add multiple transport mechanisms by using bitwise operators.

By default, the client would support all three transport mechanisms and WebSocket will be prioritized.

headers

This option allows you to apply any custom HTTP headers to your request in a form of key-value pairs.

accessTokenFactory

This setting allows you to apply authentication token. We will cover it in more detail in **chapter 8**.

logMessageContent

If set, this setting will log the content of the messages into the console.

skipNegotiation

If WebSocket transport is used, the negotiation can be skipped. Otherwise, this option will have no effect.

Skipping negotiation will establish the connection faster.

withCredentials

If set, this option will apply credentials to the request. This is especially relevant to CORS requests.

timeout

This setting determines the maximum allowed time for HTTP requests. It's not applied to poll requests of long polling, EventSource, or WebSocket.

Setting logging level

`configureLogging` method allows you to set the minimal severity level at which messages will be logged. It can be either represented by a literal string value, or a constant. When a particular severity level is set, events of that severity level and above are logged, but events of lower severity levels aren't logged.

The available severity levels are as follows:

Literal	Constant	Description
`trace`	`LogLevel.Trace`	Includes verbose output of routine processes
`debug`	`LogLevel.Debug`	Includes information intended only to be viewed during debugging
`info` or `information`	`LogLevel.Information`	Includes information on key events
`warn` or `warning`	`LogLevel.Warning`	Includes some events that may be a cause of concern
`error`	`LogLevel.Error`	Includes errors
`critical`	`LogLevel.Critical`	Includes critical failures
`none`	`LogLevel.None`	No logging is done

Next, we will apply some configuration options that may change at any time after the connection object has been created.

Changeable options

After `connection` object has been initialized, we can add the following lines of code:

```
1 connection.serverTimeoutInMilliseconds = 30000;
2 connection.keepAliveIntervalInMilliseconds = 15000;
```

Let's now go over each one of these options.

serverTimeoutInMilliseconds

This option determines how long the client should wait for a message from the server before the connection is considered dead and `onclose` event is triggered. The value is in milliseconds and it's 30 seconds by default.

keepAliveIntervalInMilliseconds

As the server sends ping messages to the clients to keep the connection alive, so do the clients send ping messages to the server. And this option determines how frequently these messages are sent by the client. By default, this value is 15 seconds, or 15,000 milliseconds.

And this concludes our overview of JavaScript client configuration. Let's now move on to .NET client.

Configuring .NET client

Before we start, we need to make sure all of the following namespace references are present in `Program.cs` file of `DotnetClient` project:

```
using Microsoft.AspNetCore.Connections;
using Microsoft.AspNetCore.Http.Connections;
using Microsoft.AspNetCore.SignalR.Client;
using Microsoft.Extensions.Logging;
```

Then, we will locate the initialization statement for `hubConnection` object and replace it with the following:

```
var hubConnection = new HubConnectionBuilder()
    .WithUrl(url,
        HttpTransportType.WebSockets,
        options => {
            options.AccessTokenProvider = null;
            options.HttpMessageHandlerFactory = null;
            options.Headers["CustomData"] = "value";
            options.SkipNegotiation = true;
            options.ApplicationMaxBufferSize = 1_000_000;
            options.ClientCertificates = new System.Security.Cryptography.X509Certif\
icates.X509CertificateCollection();
            options.CloseTimeout = TimeSpan.FromSeconds(5);
            options.Cookies = new System.Net.CookieContainer();
            options.DefaultTransferFormat = TransferFormat.Text;
            options.Credentials = null;
            options.Proxy = null;
            options.UseDefaultCredentials = true;
            options.TransportMaxBufferSize = 1_000_000;
            options.WebSocketConfiguration = null;
            options.WebSocketFactory = null;
    })
```

```
22      .ConfigureLogging(logging => {
23          logging.SetMinimumLevel(LogLevel.Information);
24          logging.AddConsole();
25      })
26      .Build();
```

You may have noticed that we have applied `HttpTransportType.WebSockets` as one of the call parameters. This is one of the ways of applying allowed transport mechanisms to .NET clients. Another way is to set it as one of the options. But whichever way you choose, you can still use bitwise operations to apply multiple transport mechanisms.

As you can see, there are more configuration options on .NET client compared to the JavaScript one. But, as before, we will overview them all.

AccessTokenProvider

This setting determines the provider for authorization token. It's used when you are applying authentication to the SignalR connection. We will cover this in more detail in **chapter 8**.

HttpMessageHandlerFactory

We can use this setting to apply a custom message handler logic to the HTTP middleware. We would need it if we would want to transform the default data in any way. Otherwise, we can just use the default value.

Headers

This option allows us to apply custom HTTP headers to the SignalR connection. Those are represented as key-value pairs.

SkipNegotiation

This option will only be applied if you are using WebSocket transport mechanism. If set to `true`, the connection will get established faster.

ApplicationMaxBufferSize

This setting represents the maximum buffer size, i.e. how much data can be transferred at once. The value is in bytes and the default value is one megabyte.

ClientCertificates

One of the common ways to authenticate a client, especially if the client is to use a certificate with stored credentials. This is especially common when the client is an automated service rather than something that an end-user would use directly. Remote procedure calls within a distributed application that uses microservices architecture is a common use case for client certificates. And this is what this option allows you to configure.

CloseTimeout

This setting controls how long the client should wait for the service response when the client has issued a request to close. If the server doesn't respond within this time, the connection is forcefully terminated.

Cookies

This option allows you to set a collection of cookies to send to the server.

DefaultTransferFormat

This option determines whether HTTP connection is started with either text or binary as the transfer format.

Credentials

This setting allows you to configure the credentials to include in HTTP request.

Proxy

This option allows you to configure a proxy that is positioned between the client and the SignalR hub.

UseDefaultCredentials

If this value is set, then default credentials will be used while making a HTTP request. This setting is applied if you want the users of Windows machines to re-apply their Microsoft Active Directory credentials to the SignalR connection.

WebSocketConfiguration

This option allows you to apply custom configuration if WebSocket transport mechanism is used.

WebSocketFactory

This delegate allows you to modify or replace the default WebSocket implementation.

Setting logging level

In C#, logging is configured via `ConfigureLogging` method call. This is where you set the log level and apply any specific logging provider. Otherwise, the available logging levels would be the same as have been described in the section dedicated to JavaScript client.

And now we will move on to those configuration options that can be changed after the connection object has been instantiated.

Applying dynamic configuration options

After we have initialized our `hubConnection` object, we can insert the following code:

```
hubConnection.HandshakeTimeout = TimeSpan.FromSeconds(15);
hubConnection.ServerTimeout = TimeSpan.FromSeconds(30);
hubConnection.KeepAliveInterval = TimeSpan.FromSeconds(10);
```

These are the options that we can change dynamically. And this is what each one of them represents:

HandshakeTimeout

This option determines how long the client should wait for the server response during the handshake before it the client terminates the connection. The default value is 15 seconds.

ServerTimeout

This determines the interval during which the client waits for the server to send a message. If no message is sent by the server during this interval, the client terminates the connection. The default value is 30 seconds.

KeepAliveInterval

This setting determines how frequently pings are sent from the client to the server. These pings will keep the connection active. And the default interval is 15 seconds.

This concludes the overview of .NET client configuration. Let's now have a look at Java client.

Configuring Java client

For Java client, logging is configured via the following package:

```
org.slf4j:slf4j-jdk14
```

If you don't have this package installed, some errors will appear in your application's console. But those will not prevent your application from working.

We will now apply some configuration to our Java client that we have created in **chapter 4**. And the first thing we will do is open our `App.java` file and replace the instantiation of `hubConnection` object with the following:

```
HubConnection hubConnection = HubConnectionBuilder.create(input)
        .withHeader("Key", "value")
        .shouldSkipNegotiate(true)
        .withHandshakeResponseTimeout(30*1000)
        .withTransport(TransportEnum.WEBSOCKETS)
        .build();
```

These are the configuration options that are available to you when a SignalR connection from Java client gets instantiated. And here is what each of these options means:

withHeader

This option determines custom HTTP headers that get applied when the connection gets established. Each of these calls represents a key-value pair.

shouldSkipNegotiate

If you are using WebSocket transport mechanism, then this option would allow you to establish the connection faster. Otherwise, it will be ignored.

withHandshakeResponseTimeout

This option sets the time for the client to wait for the server response during the handshake. If the response doesn't arrive within this time, the connection is terminated. The value is represented in milliseconds.

withTransport

This setting represents the transport mechanisms that the client will support. If not set, all three transport mechanisms are supported and WebSocket is given a priority.

Now, we will move on to those options that can be configured once the connection object is instantiated.

Dynamic configuration options

After we have instantiated our `hubConnection` object, we can add the following code.

```
hubConnection.setServerTimeout(30000);
hubConnection.withHandshakeResponseTimeout(15000);
hubConnection.setKeepAliveInterval(15000);
```

And here is what each of these options represents:

getServerTimeout / setServerTimeout

Gets or sets the interval during which the client waits for the server to send a message. If no message is sent by the server during this interval, the client terminates the connection. The value is in milliseconds. And the default value is 30 seconds.

withHandshakeResponseTimeout

This option determines how long the client should wait for the server response during the handshake before it the client terminates the connection. The value is in milliseconds. And the default value is 15 seconds.

getKeepAliveInterval / setKeepAliveInterval

This setting determines how frequently pings are sent from the client to the server. These pings will keep the connection active. The value is in milliseconds. And the default interval is 15 seconds.

This completes our overview of how to configure all types of supported SignalR clients. And now we will move on to a topic that deserves a section of its own. You will now learn how to fine-tune MessagePack protocol in SignalR.

Pros and cons of MessagePack protocol

MessagePack protocol is not exclusive to SignalR. It's similar to JSON in its structure, but it's binary rather than textual. So, you will have the same types of fields and the same object structure, but you won't be able to easily read the message while it's being transferred. But at the same time, the fact that the message is binary makes it much smaller; therefore it will get transferred faster.

Enabling MessagePack on the server

MessagePack is not included in the default SignalR packages. To enable it on the server, you will need to install a NuGet package, which can be achieved by running the following command inside `SignalRServer` project folder:

```
dotnet add package Microsoft.AspNetCore.SignalR.Protocols.MessagePack
```

Then we will open `Program.cs` file inside the project and will add the following namespace reference:

```
using MessagePack;
```

After this, we will locate `AddSignalR` method call. We can then either replace `AddJsonProtocol` or add the following call to it:

```
.AddMessagePackProtocol(options =>
{
    options.SerializerOptions = MessagePackSerializerOptions.Standard
        .WithSecurity(MessagePackSecurity.UntrustedData)
        .WithCompression(MessagePackCompression.Lz4Block)
        .WithAllowAssemblyVersionMismatch(true)
        .WithOldSpec()
        .WithOmitAssemblyVersion(true);
});
```

So, this is how we enable MessagePack on the server. We don't have to change any settings, as the default settings would be sufficient for most of scenarios. But this code demonstrates what settings are available if we do need to fine-tune it. Let's go through them all.

SerializerOptions

This setting allows you to apply custom serialization logic to your MessagePack messages. It's represented by `MessagePackSerializerOptions` class from `MessagePack` namespace, which you can override. In the above example, we are just using the standard serialization settings.

WithSecurity

This setting allows you to apply security options to MessagePack messages. It's represented by overridable `MessagePackSecurity` class from `MessagePack` namespace. This class also has `UntrustedData` and `TrustedData` static fields, which will give you pre-configured settings that would either allow or disallow deserialization of untrusted message sequences.

WithCompression

This setting allows you to apply a specific compression algorithm to your MessagePack messages. It's represented by `MessagePackCompression` enum from `MessagePack` namespace. The available values are as follows:

- `None` - no compression is applied
- `Lz4Block` - compression is applied to the entire MessagePack sequence as a single block
- `Lz4BlockArray` - compression is applied as an array of blocks, which makes compression/decompression faster, but this happens at the expense of the compression ratio

WithAllowAssemblyVersionMismatch

If this setting is set to `true`, then the assembly version, if included in MessagePack message, is allowed to be different from the one that the server is using.

WithOldSpec

If this setting is set to `true`, then old MessagePack specifications are accepted.

WithOmitAssemblyVersion

If this setting is set to `true`, then the assembly version is not included in the messages.

This concludes our overview of applying MessagePack configuration on the server. We will now have a look at how to enable it on the clients. And the first client we will have a look at will be JavaScript client.

Applying MessagePack on JavaScript client

For JavaScript client, there wasn't a suitable SignalR MessagePack library available via CDN at the time of writing. So the best way to obtain such a library is to install it via NPM.

If you haven't done it already, you will need to first initiate a Node.js project in any directory of our choice by running the following command:

```
npm init
```

Then, we will run the following command to install the official SignalR MessagePack protocol library:

```
npm install @microsoft/signalr-protocol-msgpack
```

This will create `node_modules` folder inside the folder that we are in. We will open this folder and navigate to the following location:

```
@microsoft\signalr-protocol-msgpack\dist\browser
```

Inside this folder, we will copy either `signalr-protocol-msgpack.js` or `signalr-protocol-msgpack.min.js`. It doesn't matter which one. Functionally, they are identical. The former file has human-readable content, but it's larger in size. The latter file is miniated, which means that all unnecessary characters have been removed from it and all local variable names have been shortened. It's smaller in size, but it's impossible to read.

Whichever file we have chosen, we will then navigate to `wwwroot/lib` directory inside `SignalRServer` project folder, create `signalr` folder inside this directory and will paste the JavaScript file there.

Then, we will need to add a `script` element to the `_Layout.cshtml` file. For example, if it's `signalr-protocol-msgpack.min.js` file that we have copied, the `script` element would look like the one below:

```
<script src="~/lib/signalr/signalr-protocol-msgpack.min.js"></script>
```

We will need to insert it immediately below the reference to the main SignalR library.

To add MessagePack protocol to our JavaScript client, we will then need to open `site.js` file and add the following line to the statement that builds `connection` object. This line needs to be inserted before `build` call.

```
.withHubProtocol(new signalR.protocols.msgpack.MessagePackHubProtocol())
```

And this is it. Our JavaScript client will now be using MessagePack protocol instead of JSON. We can now move on to our .NET client.

Applying MessagePack on .NET client

To enable MessagePack protocol on a .NET client, you would need to install exactly the same NuGet package as you would on the server. So, we would execute the following command in our `DotnetClient` project folder:

```
dotnet add package Microsoft.AspNetCore.SignalR.Protocols.MessagePack
```

Then we will need to ensure that `Program.cs` file in the `DotnetClient` project has all of the following namespace references:

```
using MessagePack;
using Microsoft.AspNetCore.Connections;
using Microsoft.AspNetCore.Http.Connections;
using Microsoft.AspNetCore.SignalR.Client;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using System.Threading.Channels;
```

Then we can navigate to the place where `hubConnection` variable is initialized and add the following block of code to the builder immediately before `Build` call:

```
.AddMessagePackProtocol(options =>
{
    options.SerializerOptions = MessagePackSerializerOptions.Standard
    .WithSecurity(MessagePackSecurity.UntrustedData)
    .WithCompression(MessagePackCompression.Lz4Block)
    .WithAllowAssemblyVersionMismatch(true)
    .WithOldSpec()
    .WithOmitAssemblyVersion(true);
})
```

As you may have noticed, .NET client MessagePack settings are identical to the ones available on the server. And this is because both of them use the same library.

We will now configure our Java client to use MessagePack protocol.

Applying MessagePack on Java client

SignalR MessagePack package for Java can be found on the following page of Maven repository:

https://mvnrepository.com/artifact/com.microsoft.signalr.messagepack/signalr-messagepack

You will need to select the latest version and then just follow the installation instruction that is appropriate to the build system that you are using in your application. For example, assuming that the latest version is 6.0.1, if you are using Maven, you would need to add the following block into the `dependencies` section of `pom.xml` file:

```
<dependency>
    <groupId>com.microsoft.signalr.messagepack</groupId>
    <artifactId>signalr-messagepack</artifactId>
    <version>6.0.1</version>
</dependency>
```

If you are using Gradle with Groovy DSL, this is the entry you would need to insert into `dependencies` section of `gradle.build` file:

```
implementation 'com.microsoft.signalr.messagepack:signalr-messagepack:6.0.1'
```

This would be the entry you would need to insert into `dependencies` section of `gradle.build.kts` file if you have used Kotlin DSL during project setup:

```
implementation("com.microsoft.signalr.messagepack:signalr-messagepack:6.0.1")
```

Then, once this package has been added, you can apply MessagePack protocol to your SignalR connection object by adding `.withHubProtocol(new MessagePackHubProtocol())` line on `HubConnectionBuilder` before `build` is called. For example, this what the instantiation of connection object may look like:

```
HubConnection hubConnection = HubConnectionBuilder.create(input)
        .withHubProtocol(new MessagePackHubProtocol())
        .build();
```

And this concludes MessagePack protocol setup on a Java client.

But as well as having its advantages in terms of decreased payload size, MessagePack protocol has some disadvantages. We will now examine those.

Disadvantages of MessagePack protocol

The first disadvantage of MessagePack protocol is that it's not available out of the box. You need to install external packages to enable it.

Another disadvantage of MessagePack protocol compared to JSON is that it's highly case-sensitive. While JSON easily translates between camelCase JavaScript object field names and PascalCase C# property names, MessagePack requires the names to match exactly. For example, if you use the following class in C# as one of SignalR hub method parameters:

```
public class Data
{
    public int Id { get; set; }
    public string Message { get; set; }
}
```

Then this is how JavaScript client will have to invoke the method that uses this message:

```
connection.invoke("SomeHubMethod", { Id: 1, Message: "Some text" });
```

This will go against the accepted naming conventions in JavaScript. And if you already have a lot of JSON data in your JavaScript client before you've decided to switch to MesssagePack hub protocol, the migration process would not be as straight-forward.

There is a work-around available. You can use `Key` attribute on your C# properties to map the names to camelCase versions of them. But it still requires some effort, especially if you have a lot of classes in your code that represent SignalR messages. For more information on how field mapping works, you can read `MessagePack-CSharp` documentation, which is available in **further reading** section of this chapter.

We have now concluded the overview of advanced SignalR configuration. Let's summarize what we've learned.

Summary

SignalR server has three levels of configuration: top-level SignalR configuration, message protocol configuration and HTTP transport configuration. You can apply custom serializers, change default timeouts and buffer sizes, change transport mechanisms, etc.

Different SignalR client types have different configuration options available. But all clients support configuration of transport mechanism (WebSocket, server-sent events and long polling), timeouts and logging.

By default, clients communicate with SignalR hub by using JSON. But MessagePack protocol can be used instead. This protocol is similar to JSON, but stores data in a binary format. Although it's better for performance, it's more strict than JSON and you will need to install additional libraries to use it.

And this concludes the chapter about configuring SignalR. The next chapter will teach you how to secure your SignalR endpoints and prevent unauthorized clients from accessing them.

Test yourself

1. Under which circumstances does it make sense to pass detailed server errors to the client?
   A. In a production-grade application
   B. In a development environment
   C. In an environment dedicated to user acceptance testing
   D. All of the above
2. Where can you specify which transport mechanisms would SignalR be allowed to use?
   A. In the server configuration
   B. In the client configuration
   C. In either the server or the client configuration
   D. You cannot explicitly specify transport mechanisms in SignalR configuration
3. What are the disadvantages of MessagePack protocol?
   A. You need to install additional packages to use it
   B. It has stricter rules than JSON
   C. All of the above
   D. None of these

Further reading

Official SignalR configuration documentation: https://docs.microsoft.com/en-us/aspnet/core/signalr/configuration

MessagePack protocol in SignalR: https://docs.microsoft.com/en-us/aspnet/core/signalr/messagepackhubprotocol

MessagePack-CSharp library documentation: https://github.com/neuecc/MessagePack-CSharp

General MessagePack overview: https://msgpack.org/

WebSocket API documentation: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

8 - Securing your SignalR applications

Securing your applications is very important, especially if they are accessible via a network that you don't have control over, such as public internet. Without security, absolutely anyone can connect to your application, including someone who is up to no good. And this may result in a catastrophic failure that your business may not then be able to recover from.

In this chapter, we will have a look at how to protect your SignalR endpoints. We will start by looking at the way of restricting your application to only be accessible by clients from specific domains. And then we will set up a single sign-on system, which will allow us to apply user authentication and authorization on SignalR endpoints. The chapter will explain the difference between authentication and authorization, and you will learn how to integrate them both with SignalR.

The chapter consists of the following topics:

- What is CORS and why it's important
- Setting up single sign-on provider
- Applying authentication in SignalR
- Applying authorization in SignalR

By the end of this chapter, you will have learned how to make your SignalR endpoints secure by preventing unauthorized clients from accessing them. You will learn how to secure individual hub endpoints as well as the entire SignalR hub.

Prerequisites

This chapter assumes that you already have set up your development environment, as described in **chapter 1**. You will need the following:

- A machine with either Windows, Mac OS or Linux operating system
- A suitable IDE or code editor (Visual Studio, JetBrains Rider or VS Code)
- .NET 6 SDK (or newer)

Also, since we are continuing to build on top of the application that we have developed in the previous chapter, we need the code that we have written previously. If you have skipped the previous chapter, you can access the complete code from the following location in the GitHub repository:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide/tree/main/Chapter-07/Part-03/LearningSignalR

The complete code samples from this chapter are available from the following location in the GitHub repo, which has separate folders corresponding to individual parts of the chapter:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide/tree/main/Chapter-08

What is CORS and why it's important

Perhaps one of the most basic things you can do to your ASP.NET Core application to make it more secure is to enable CORS policies. CORS stands for Cross-Origin Resource Sharing. And it's all about allowing (or restricting) access to the client applications hosted on specific domains that the clients are hosted on.

CORS configuration doesn't just apply to SignalR. It applies to the entire ASP.NET Core application and all of its HTTP endpoints, including SignalR.

Without further ado, let's add some CORS policies to our application. And to do so, we will insert the following into `Program.cs` file of `SignalRServer` project. It can go anywhere before `Build` method is called on `builder` variable.

```
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowAnyGet",
        builder => builder.AllowAnyOrigin()
            .WithMethods("GET")
            .AllowAnyHeader());

    options.AddPolicy("AllowExampleDomain",
        builder => builder.WithOrigins("https://example.com")
            .AllowAnyMethod()
            .AllowAnyHeader()
            .AllowCredentials());
});
```

Let's now examine what we've done. We have added two CORS policies. One is called `AllowAnyGet` and the other one is called `AllowExampleDomain`. Inside `AllowAnyGet` policy, we are allowing any HTTP request from any domain, but only if it uses `GET` HTTP verb. `AllowExampleDomain` policy allows any method with any header. It also allows the application to accept credentials. But it only allows those things if a request comes from `https://example.com` origin.

And now, we can apply the policies that we have added. To do so, we will insert the following code just after the call to `app.UseRouting`:

```
app.UseCors("AllowAnyGet")
    .UseCors("AllowExampleDomain");
```

Please note that it is important in what order you call methods on `app` variable. After all, these are request middleware stages. Therefore the capabilities that are represented by these method calls are added to the request processing chain in the same order as these methods are called. For `UseCors` method, the order is less critical than it is for some other steps. But if you have `UseResponseCaching` method call in your middleware pipeline, `UseCors` must be called first.

You will probably see no effect of this configuration with our client applications, unless you move any of the stand-alone clients to a different machine. But this exercise was still useful, as you now know how to blacklist or whitelist domains that can access your application.

And now we will move on to a different method of securing our SignalR server endpoints. We will set up a single sign-on system (SSO).

Setting up single sign-on provider

SSO is a system where different applications can re-use the same authentication data. Usually, there will be a single application within this system that manages user login information. We will refer to such an application as SSO provider. This application will generate authentication data that can then be used by other applications and shared between them. So, if you logged into one application of the system, you have automatically logged into them all.

There are many different SSO providers: Keycloak, Okta, Microsoft Azure Active Directory, etc. Major tech companies use their own SSO providers. This is what allows you to log on a website that you've never used before by using your Google or Facebook credentials.

The compatibility of apps with different SSO providers was made possible by the standardization of authentication protocols. More often than not, those system would be using a combination of OpenID Connect and OAuth. Let's briefly have a look at what those are.

Overview of OpenID Connect and OAuth

OpenID Connect is a protocol that was specifically designed for authentication, while OAuth is authorization protocol. And this is how these two protocols can be used in combination. OpenID Connect defines the login process. And OAuth determines the structure of authentication token that will allow the system to easily tell if the user has all the required permissions to access a particular resource.

There are several different authentication flows in OpenID Connect, but the general principles of them all can be summarized as follows:

1. A user opens an application and initiates a login

2. The application gets redirected to the authentication page of SSO provider application
3. If the user enters credentials correctly, a time-sensitive code is generated by the SSO application
4. SSO provider redirects the user back to the original application and shares the code with this application
5. The code is sent back to SSO provider and, if the codes match, authentication token is returned to the original application
6. The token can be stored in a cookie, so the application knows that the user is authenticated

Please note that the original application that the user wants to access has absolutely no idea what the user credentials are. For example, when you use "Log in with Google" option on the web, you get taken to Google's own page. And it's that page where you enter the credentials. You return back to the original page only if the credentials you supplied were correct. And to enable this, a redirection URL is sent to the SSO provider when the log in page is requested. So rest assured that you won't be sharing your Google or Facebook credentials with some third-party website.

The authentication token that gets retrieved from the SSO provider usually comes as JSON Web Token (JWT). We will have a look at JWT structure later. For now, let's set up an SSO provider of our own.

In our example, we will be using IdentityServer. This specific SSO provider was chosen because it's based on ASP.NET Core project template, so it will be easy to add to our solution.

Setting up IdentityServer 4

We will need to open our CLI terminal in `LearningSignalR` solution folder. Then, we will need to execute the following command to download IdentityServer 4 project templates:

```
dotnet new -i IdentityServer4.Templates
```

Once the templates have been downloaded, we will execute the following command to instantiate a project based on a template that contains Admin UI. We could have chosen a different template, but this particular one has a user-friendly web interface, so it will be more convenient to work with than any alternative.

```
dotnet new is4admin -o AuthProvider
```

Then, we will add our newly created project to the solution by executing the following command:

```
dotnet sln add AuthProvider\AuthProvider.csproj
```

By default, IdentityServer 4 comes with only a limited set of its capabilities unlocked. We will need to make some custom code modifications to ensure that all the information that we are interested in gets passed into the authentication token. And to do so, we will create `UserProfileService.cs` file inside `AuthProvider` project folder. We will add the following content to this file:

```
using AuthProvider.Models;
using IdentityModel;
using IdentityServer4.Extensions;
using IdentityServer4.Models;
using IdentityServer4.Services;
using Microsoft.AspNetCore.Identity;
using System;
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;

namespace AuthProvider
{
    public class UserProfileService : IProfileService
    {
        private readonly IUserClaimsPrincipalFactory<ApplicationUser> claimsFactory;
        private readonly UserManager<ApplicationUser> usersManager;

        public UserProfileService(
            UserManager<ApplicationUser> usersManager,
            IUserClaimsPrincipalFactory<ApplicationUser> claimsFactory)
        {
            this.usersManager = usersManager;
            this.claimsFactory = claimsFactory;
        }
    }
}
```

The class that we have created implements `IProfileService` interface from `IdentityServer4.Services` namespace. This interface allows us to execute some custom logic when an authentication token gets requested. And this is the method that we will add to it to execute such logic:

```
public async Task GetProfileDataAsync(ProfileDataRequestContext context)
{
    var subject = context.Subject.GetSubjectId();
    var user = await usersManager.FindByIdAsync(subject);
    var claimsPrincipal = await claimsFactory.CreateAsync(user);
    var claimsList = claimsPrincipal.Claims.ToList();

    claimsList = claimsList
        .Where(c => context.RequestedClaimTypes
        .Contains(c.Type))
        .ToList();
```

```
12
13      // Add user-specific claims
14      claimsList.Add(new Claim(JwtClaimTypes.Email, user.Email));
15      claimsList.Add(new Claim(JwtClaimTypes.Name, user.UserName));
16
17      if (usersManager.SupportsUserRole)
18      {
19          foreach (var roleName in await usersManager.GetRolesAsync(user))
20          {
21               claimsList.Add(new
22              Claim(JwtClaimTypes.Role, roleName));
23
24              // Add a special claim for an admin user
25              if (roleName == "admin")
26                  claimsList.Add(new Claim("admin", "true"));
27          }
28      }
29      context.IssuedClaims = claimsList;
30  }
```

Let's examine what this method is doing. The payload of JWT would be a JSON object with multiple fields. Each of these fields is known as *claim*. And we are just adding some claims that were missing from the original payload.

We first add some claims specific to the user. We populate the `email` claim with the user's email and we add the username to the `name` claim. Then we iterate through all available user roles. We add each one of those to `role` claim. The result of adding multiple values to the same claim would be that the value of this JSON field will be an array. So, we will still have a claim called `role`. But inside this claim, there would be a collection of roles.

Then, if we locate a role called `admin`, we add a special custom claim to the token. We create a claim called `admin` and set its value to `true`.

We will have a look at how claims work in more detail when we will be talking about user authorization. But for now, we will need to complete our newly created service by adding the following method to it. It's not doing anything interesting, but we need it to make sure that `IProfileService` interface is implemented.

```
public async Task IsActiveAsync(IsActiveContext context)
{
    var subject = context.Subject.GetSubjectId();
    var user = await usersManager.FindByIdAsync(subject);
    context.IsActive = user != null;
}
```

Now, we will need to register our newly created class in the dependency injection system. To do so, we will need to add the following namespace reference to the `Startup.cs` file of the `AuthProvider` project:

```
using IdentityServer4.Services;
```

Then, we will need to insert the following line right at the bottom of `ConfigureServices` method. Placing right at the bottom of this method is important to guarantee that we overwrite the original implementation.

```
services.AddScoped<IProfileService, UserProfileService>();
```

Now, will need to ensure that our SSO provider application is accessible via HTTPS and not just HTTP. This is because, without any additional configuration, many frameworks and browsers would not work with an SSO provider that doesn't have HTTPS encryption.

To add HTTPS URL, we will need to open `launchSettings.json` file that's located in `Properties` folder of our SSO project. Then, we will need to replace `applicationUrl` entry with the following:

```
"applicationUrl" : "https://localhost:5001;http://localhost:5000"
```

Now, our application will be accessible via `https://localhost:5001` URL and this will be the URL that will be given priority. We just need to make sure that this is the default URL the internal components of the application would be accessible on. To do so, we will need to open `env.js` file that is located in `wwwroot/admin/assets` folder. In there, just replace every URL with `https://localhost:5001`.

Our SSO provider application is now ready to be launched and configured.

Configuring SSO application

We can launch our application by executing `dotnet run` command inside the `AuthProvider` project folder. The first launch may take a while, as it will need to run some database migrations. Once ready, we can open `https://localhost:5001/admin` address in our browser to ensure that our application works correctly.

Assuming that our admin console launches successfully and there are no errors displayed on the screen, we can register a client application. In the context of SSO, client application is any application that is allowed to redirect to the login page of the SSO application and request an authentication token from it. The configuration information that we enter in the SSO admin console and the information we add to the client application itself must match in order for this to work.

We will first navigate to **Clients** tab and create new client. In the dialog that opens, we will choose **Web Application**. as our client type, like the following picture demonstrates:

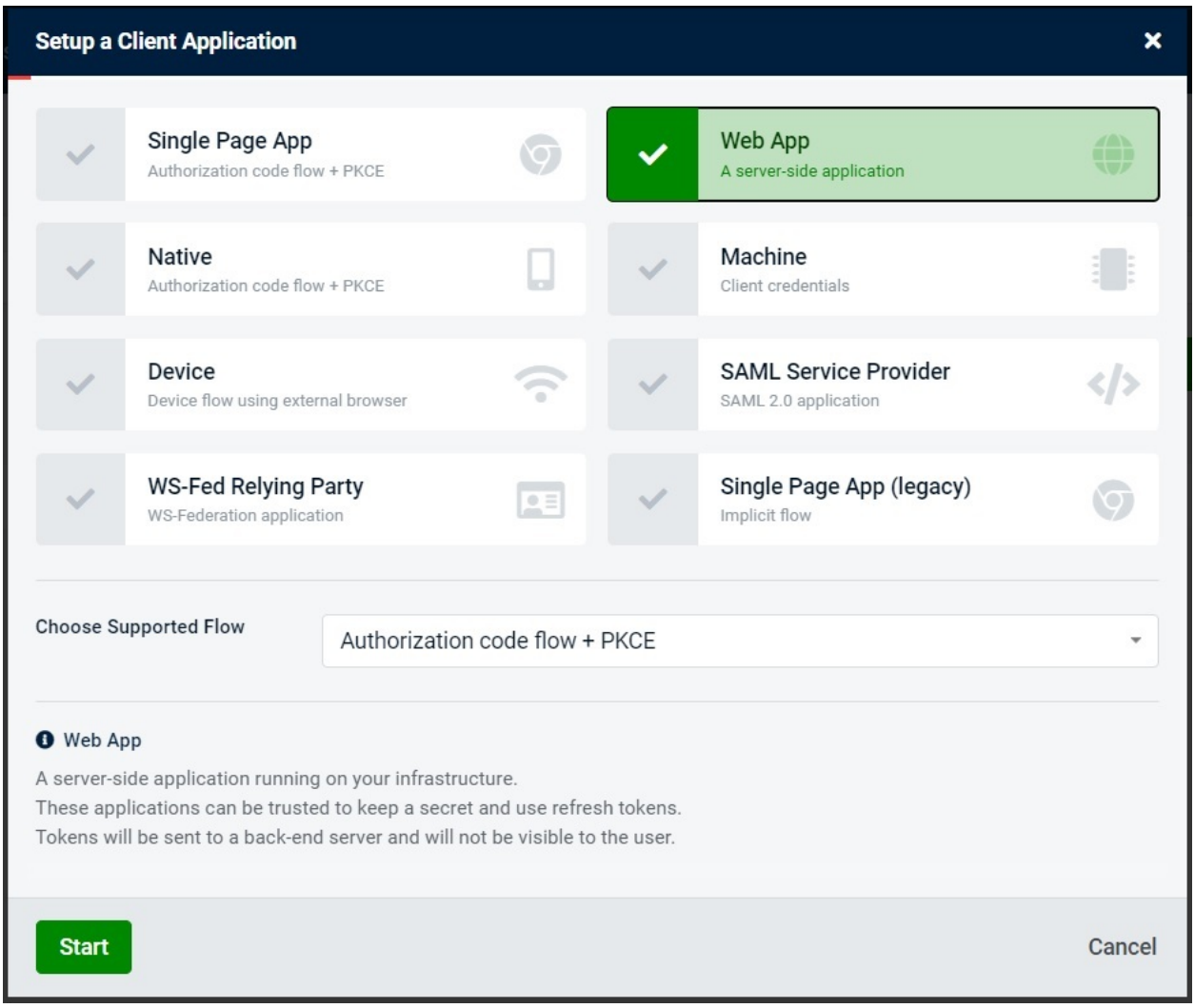

Figure 8.1 - Selecting a client application type

We will then enter `webAppClient` as both client name and client id, like shown on the screenshot below. Client name can be anything, but it's important that the client id is the same as what's configured on the actual client application, as this information will be used in the requests to SSO. And it will be used by the SSO provider to authenticate the client application and prevent malicious users from sending a request from somewhere else.

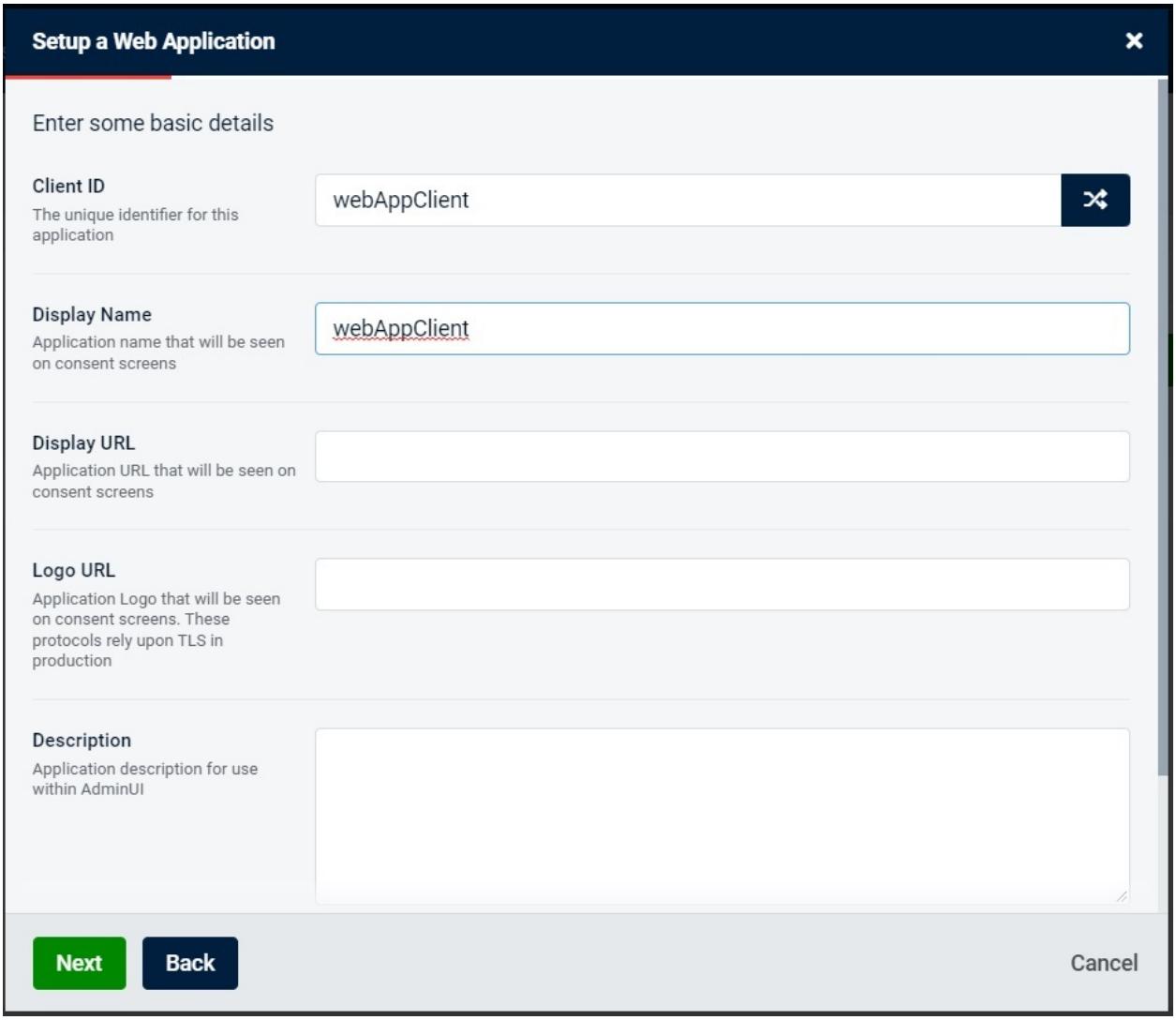

Figure 8.2 - Filling up the details of the client

Next, we will need to add the redirection URL. This is the URL that the SSO provider will redirect to if the user is successfully authenticated. This can be overridden by the client, which can send a different endpoint in the request.

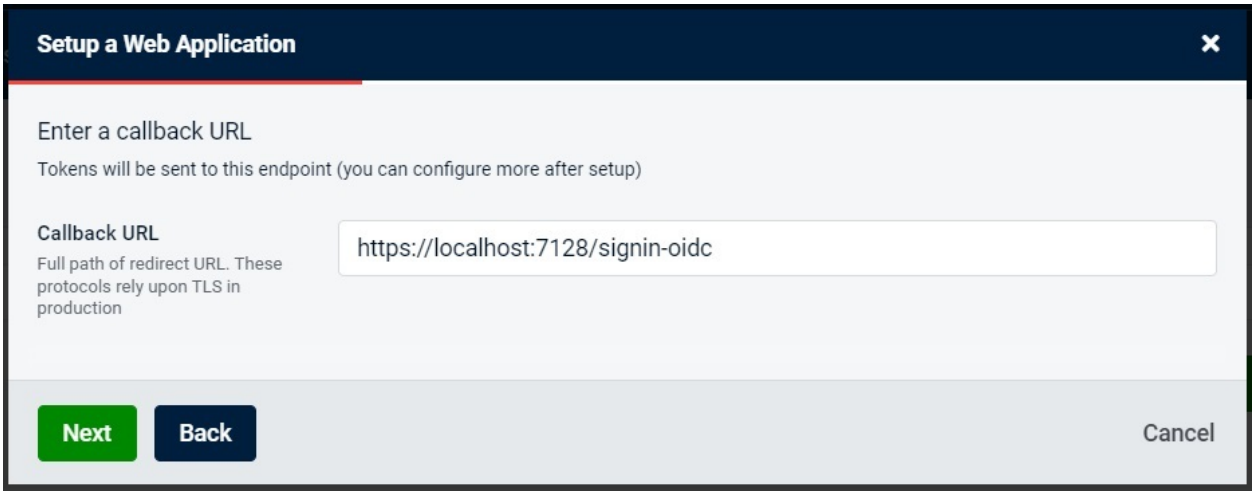

Figure 8.3 - Setting up redirect URL

Next, we will need to add a client application secret. This value would be hidden and normally it would be represented by a complex string of characters. But for the sake of demonstration, we will set this value to `webAppClientSecret`. If client id is analogous to the username, then client secret is an equivalent of the password.

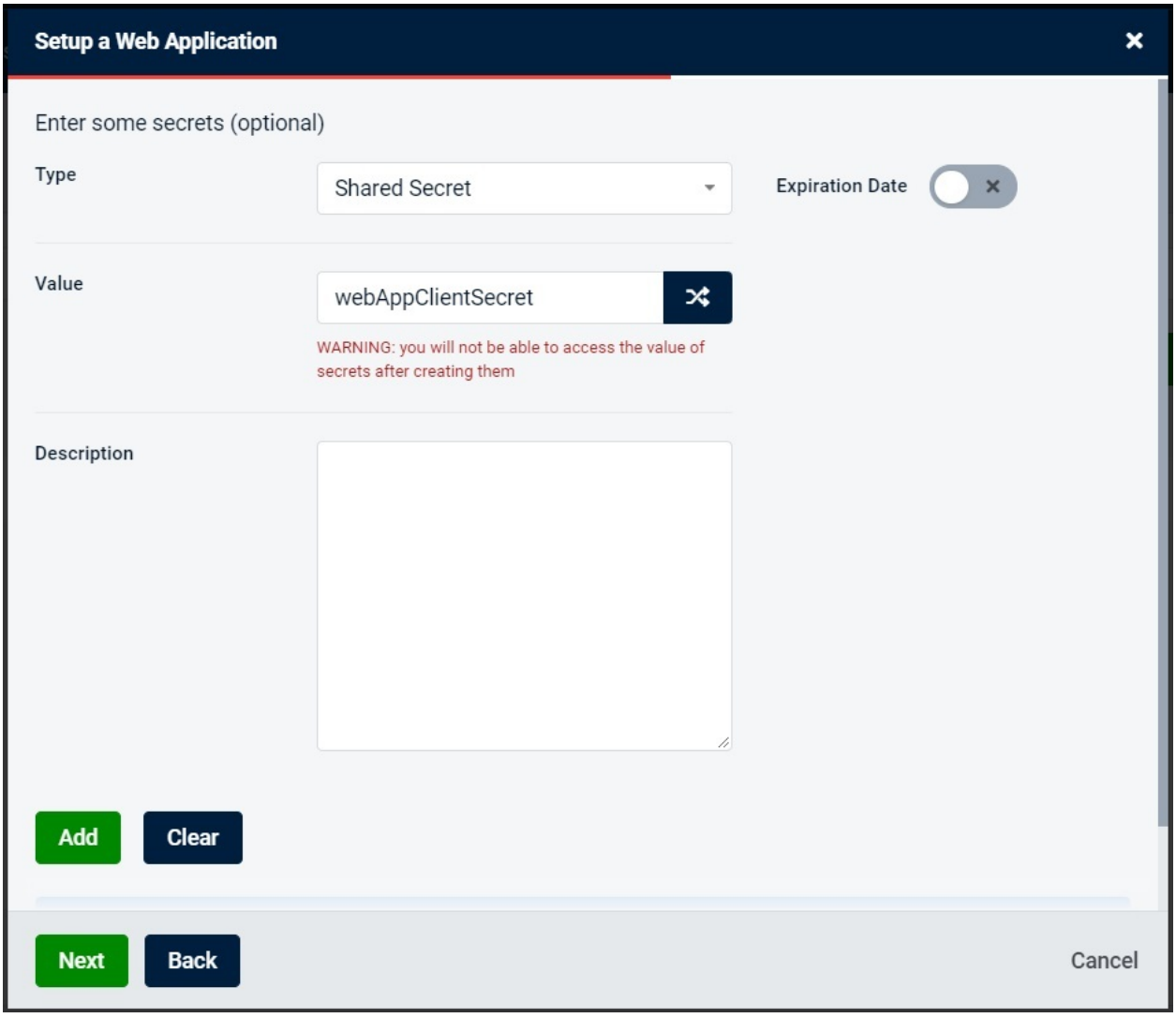

Figure 8.4 - Setting up client application secret

Next, we will add some client scopes, as demonstrated on the screenshot below. And we will just select default values in all the following steps.

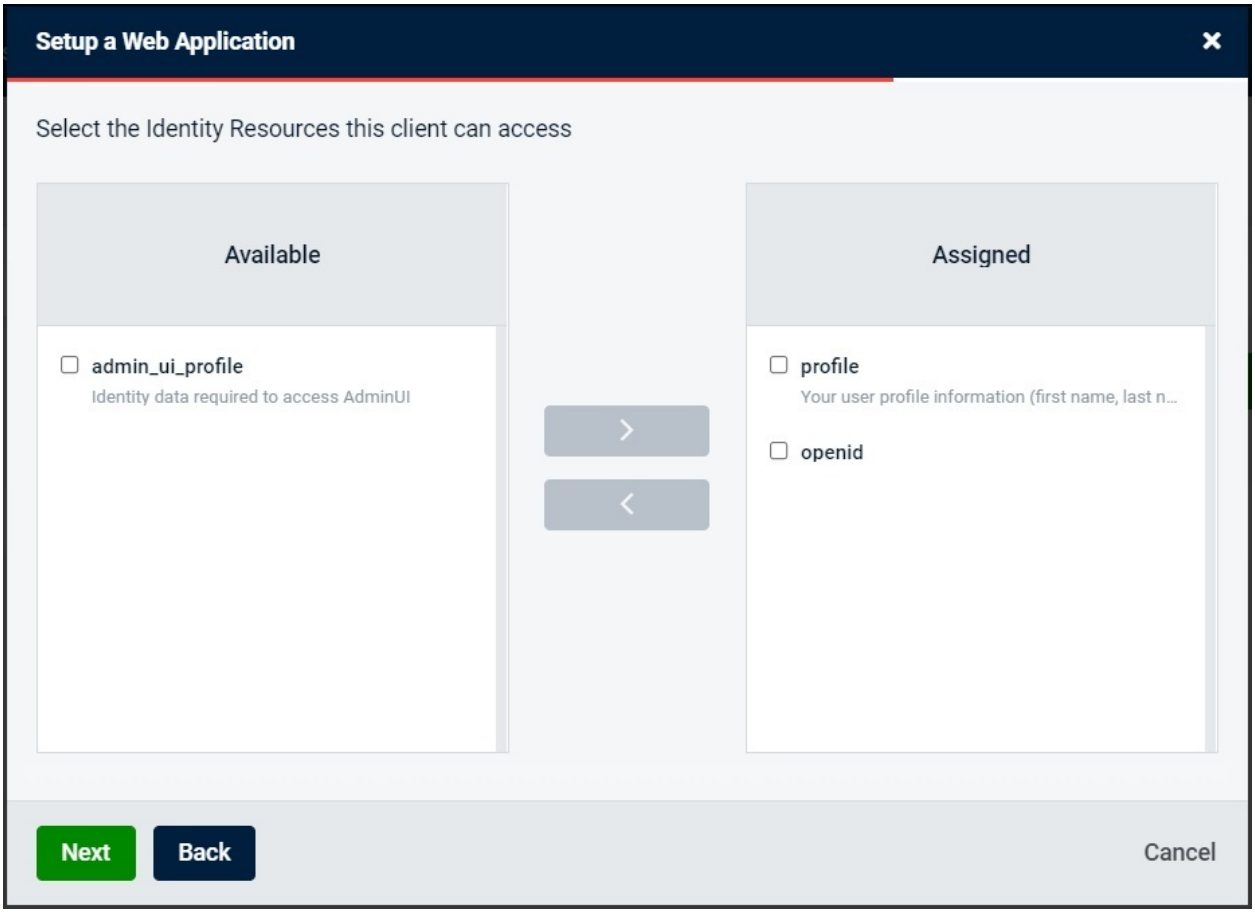

Figure 8.5 - Adding client scopes

Once the client has been created, we will navigate to **Roles** tab. We will add two roles - `user` and `admin`.

Next, we will navigate to the **Users** tab and create some users. Once a user has been created, we can navigate to the **Roles** sub-tab of the user details section, and we can add roles to the user by selecting them in the left box and clicking on the button with the arrow pointing right, as can be seen on the following screenshot:

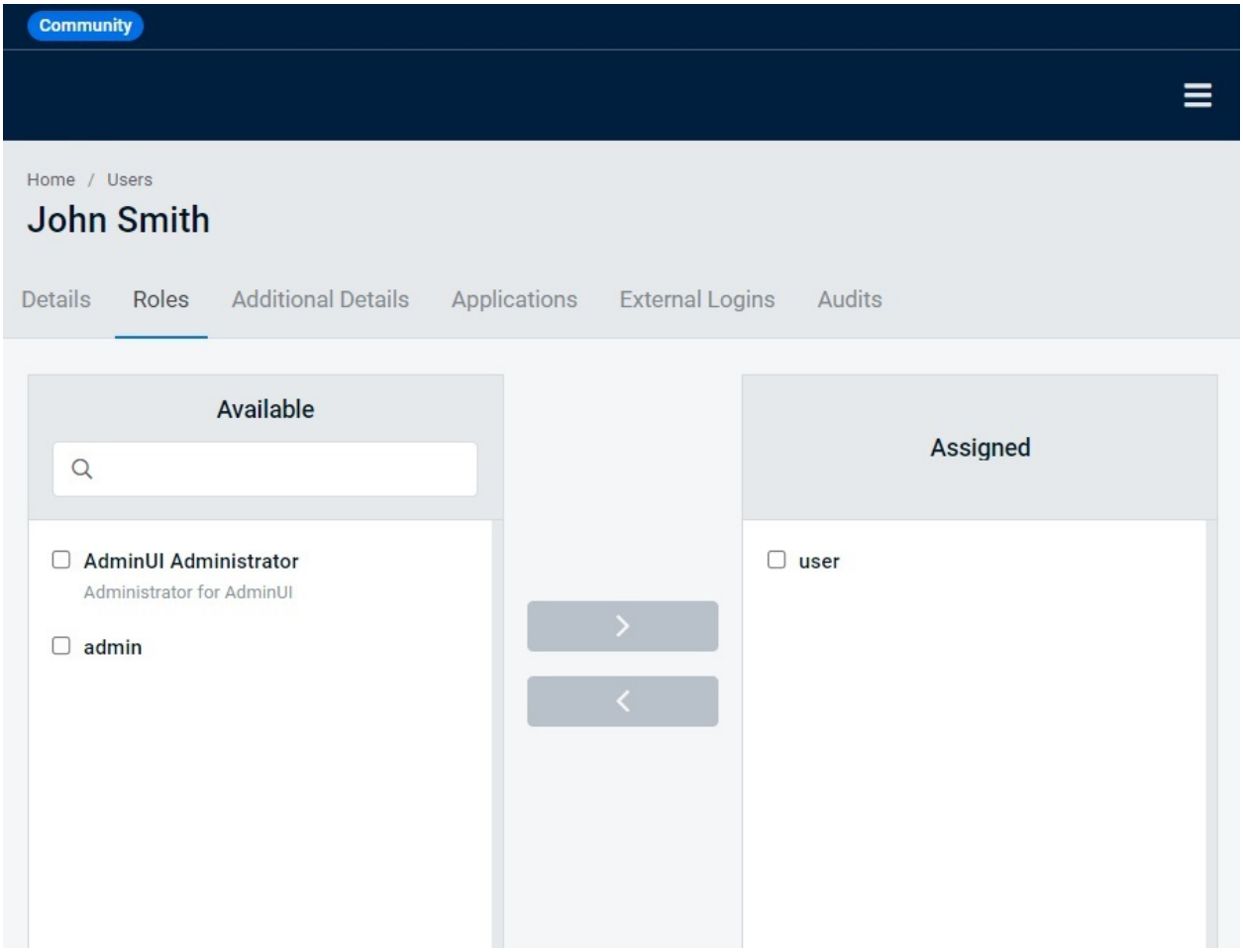

Figure 8.6 - Adding roles to the user

And now we have finished setting up our SSO provider. Next, we will modify our SignalR server to make sure that only authenticated users can connect to it.

Applying authentication in SignalR

We will first enable authentication middleware on our SignalR server application. Then we will apply access restrictions to the SignalR hub, so only authenticated users would be allowed to access it. And then we will ensure that our clients are authenticated.

Setting up authentication on SignalR server

To apply authentication to our SignalR hub, we will first need to configure and enable authentication middleware. And before we do this, we need to ensure that `JwtBearer` and `OpenIdConnect` NuGet packages have been added to our `SignalRServer` project. To add them, you can either locate and install them via NuGet package manager of your IDE, or execute the following commands inside the project folder:

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
dotnet add package Microsoft.AspNetCore.Authentication.OpenIdConnect
```

Then, we will open `Program.cs` file of the project and add the following namespace references:

```
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using System.IdentityModel.Tokens.Jwt;
```

Then, we will add configure OpenID Connect and cookie authentication middleware by adding the following code anywhere before `builder.Build` method is called:

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = "oidc";
})
.AddCookie(CookieAuthenticationDefaults.AuthenticationScheme)
.AddOpenIdConnect("oidc", options =>
{
    options.Authority = "https://localhost:5001";
    options.ClientId = "webAppClient";
    options.ClientSecret = "webAppClientSecret";
    options.ResponseType = "code";
    options.CallbackPath = "/signin-oidc";
    options.SaveTokens = true;
    options.RequireHttpsMetadata = false;
})
```

So, this is what we are doing here. We are first setting the default authentication scheme and the default challenge scheme. *oidc* stands for *OpenID Connect*, so we are just telling our middleware that this is the authentication mechanism that we are using. And we are using cookie authentication by default.

We can't use cookie authentication unless we add a handler for it. And this is precisely what `AddCookie` method does. We just add the name of the authentication scheme to it that we have set as default.

Then we configure our OpenID Connect options. And this is where we configure our client. Normally, all of these options would come from a configuration file and would be different on different environments. But to make it easier to demonstrate the principles, we have hard-coded them.

Authority

This setting contains the base URL of our SSO provider. The middleware will construct the full URL when required.

ClientId

This setting contains our client id, which must match exactly with the client id we have configured in the admin section of our IdentityServer instance.

ClientSecret

This value must match the client secret we had configured in our SSO provider application. Normally, this value would come from an encrypted source, such as Azure Key Vault, and it would not be easily readable. But we have hardcoded it here just for the sake of demonstration.

ResponseType

There are multiple ways OpenID Connect can be configured to send the response. Response type `code` represents a time-sensitive code that gets given to the client application after a user has successfully authenticated.

CallbackPath

This is the path of that is used as a redirect address. The base URL would be the URL of our current application.

SaveTokens

When this option is enabled, the application will automatically store the authentication token in a cookie. It will also store refresh tokens.

RequireHttpsMetadata

This option is needed if the SSO provider is accessible via an unencrypted HTTP address. Since our SSO provider is accessible via HTTPS, we don't need this setting. And, unless you are on a development environment, there is no chance that you would ever be dealing with an SSO provider that is not accessible via HTTPS. But we have placed this setting here just to show what it does.

So, we have configured OpenID Connect and we have added cookie authentication for our browser-based clients. But what about external apps that can't easily access the authentication page? Well, in this case, we can add an additional handler that can process a JWT bearer token. And to do so, we will just need to append the following to our call:

```
.AddJwtBearer(options =>
{
    options.Authority = "https://localhost:5001";
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateAudience = false
    };
    options.RequireHttpsMetadata = false;
    options.Events = new JwtBearerEvents
    {
        OnMessageReceived = context =>
        {
            var path = context.HttpContext.Request.Path;
            if (path.StartsWithSegments("/learningHub"))
            {
                // Attempt to get a token from a query sting used by WebSocket
                var accessToken = context.Request.Query["access_token"];

                // If not present, extract the token from Authorization header
                if (string.IsNullOrWhiteSpace(accessToken))
                {
                    accessToken = context.Request.Headers["Authorization"]
                        .ToString()
                        .Replace("Bearer ", "");
                }

                context.Token = accessToken;
            }

            return Task.CompletedTask;
        }
    };
});
```

The settings inside this handler are similar to the ones we have used inside the `AddOpenIdConnect` call. We just aren't supplying the client information used by the SSO provider, as this handler accepts the token that already has been generated. There is no redirection happening here.

We have applied `TokenValidationParameters` setting to make the token validation less strict. In our case, because our token would not contain **audience** claim that determines which specific endpoints the token would be allowed to authenticate against, we set `ValidateAudience` setting to `false`.

We also have `OnMessageReceived` event handler inside this block of code. This is the code that gets executed when the application receives a request. In most cases, the token that we will receive will

be present inside `Authorization` header. The header would contain the token type (which, in this case, would be `Bearer`) and the actual base64-encoded token value. This is why we are removing `Bearer` from the header to extract the actual token.

However, some types of requests, for example, browser-initiated WebSocket and server-sent events connections can't use `Authorization` header. Instead, the token would be sent in `access_token` query string parameter. The event handler doesn't know how the client would send the token to it, so we just check both the header and the query string. Then, once the token has been extracted, we apply it to the context object.

Please note that, by default, JWT handler would apply token verbatim. All claims from the token would be transferred into the application without any changes. Cookie authentication handler alongside OpenID Connect handler, however, has its own in-built logic when it maps the claims. Some of them may be mapped to claims with names that are different from the ones we had in the original token. This behavior can be controlled by using `JwtSecurityTokenHandler.DefaultMapInboundClaims`. For example, we can insert the following line either inside any specific authentication handler or outside the handlers to prevent the token claims from being mapped to some inbuilt names:

```
JwtSecurityTokenHandler.DefaultMapInboundClaims = false;
```

Or we can use the following line to clear the mapped claims:

```
JwtSecurityTokenHandler.DefaultMapInboundClaims.Clear();
```

Once we've set up our authentication middleware, we need to enable it. In ASP.NET Core, authentication and authorization are linked. We will be using authorization attributes for the purpose of authentication. And this is why you would need to have both of the following next to each other in your middleware pipeline in `Program.cs` file:

```
app.UseAuthentication();
app.UseAuthorization();
```

Now, we will actually apply authentication restriction on our SignalR hub. Before we do so, we will need to add the following namespace references to `LearningHub.cs` file, unless you haven't marked them as `global` elsewhere in the project code:

```
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;
```

Then, we will add the following attribute just above the class definition:

```
[Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme + "," +
CookieAuthenticationDefaults.AuthenticationScheme)]
```

This is it. Now only authenticated users can access our hub. And it's exactly the same `Authorize` attribute that we use elsewhere in ASP.NET Core applications, such as REST API controllers and gRPC services.

Normally, `Authorize` attribute could be applied without any parameters. But because we use two authentication schemes in our application, we would need to define them both. Otherwise only the default authentication scheme would work.

We can also apply this attribute to individual methods of the hub rather than the whole hub. If we decide that most of the hub needs to be restricted, but it's OK to allow a handful of its methods to be accessible by unauthenticated users, we can apply the following attribute to those methods:

```
[AllowAnonymous]
```

The authentication configuration on our SignalR server is almost finished. There is just a couple of final bits that we need to do before we can connect clients to it. We will need to print the access token into the application console once we log in, so we can copy and paste it into request headers to make it possible to authenticate from other client applications. And we need to implement some log out functionality. Both of these will be implemented inside `HomeController.cs` file. And before we start, we need to ensure that the file has the following namespace references:

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authorization;
```

Then, we will add the following code at the beginning of the `Index` method, which will retrieve the token from our HTTP context and print it in the console:

```
var accessToken = await HttpContext.GetTokenAsync("access_token");
Console.WriteLine($"Access token: {accessToken}");
```

Finally, we will add the following method, which, when triggered, will clear the authentication cookies and will log us out:

```
public IActionResult LogOut()
{
    return new SignOutResult(new[] {CookieAuthenticationDefaults.AuthenticationSchem\
e, "oidc" });
}
```

We just now need to ensure that this new endpoint is accessible from our web page. And to do so, we will open `_Layout.cshtml` file and insert the following markup inside `ul` element with the `navbar-nav` `class` attribute:

```
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" aspcontroller="Home" asp-action="LogOu\
t">Log Out</a>
</li>
```

And this is it. Only authenticated users can now access the hub. We now just need to make sure that our clients can authenticate. And this is what we will do next.

Authenticating JavaScript client

Because we are using cookie authentication, we don't need to add anything extra to our JavaScript client. The back-end configuration ensures that the page is redirected to the login page if the user is not yet authenticated. And then the authentication cookie is being passed automatically in the request.

However, if we were to use our JavaScript client in a different context, we could get it to use Bearer token for authentication instead of a cookie. To do so, we would just need to add the following option to `withUrl` call on `HubConnectionBuilder`:

```
accessTokenFactory: () => myToken
```

`myToken` represents a token that we obtain from SSO provider. Obtaining the token outside the browser is beyond the scope of this article, as it's not a subject that is specific to SignalR. However, information on it is available in the official OpenID Connect documentation, which can be found in **further reading** section of this chapter.

JavaScript client also has `withCredentials` option, which, if set to true, will apply direct credentials to the request that it can extract from a specific cookie. This is especially applicable for Azure App Service, which uses cookies for sticky sessions. This type of service will not work correctly without this option enabled.

And these are all the authentication parameters we can use in JavaScript client. Let's now move on to .NET client.

Authenticating .NET client

In the `Program.cs` file of the `DotnetClient` project, we will add the following lines after we have been prompted

```
Console.WriteLine("Please specify the access token");
var token = Console.ReadLine();
```

Then, we can replace `WithUrl` call on `HubConnectionBuilder` with the following:

```
.WithUrl(url, options => {
    options.AccessTokenProvider = () => Task.FromResult(token);
})
```

We have removed all unnecessary configuration and we can now apply JWT to the requests.

In this case, JWT that we provide is the one that we copy and paste manually. This way, the concept is easier to demonstrate. But in a real-life application, this action associated with `AccessTokenProvider` would be obtaining the token from the SSO provider.

But `AccessTokenProvider` option is not the only option we can use in our .NET SignalR client for authentication. We can use `Cookies` option to apply cookie authentication. However, it's not a recommended way to authenticate from a stand-alone client, as the cookies are harder to work with than they are in the browser. We also have `Credentials` option that allows us to send the credentials to the application rather than the token. Finally, there's `UseDefaultCredentials` option, which allows us to apply Microsoft Active Directory credentials of the logged-in user. This option is only applicable on Windows.

And now we will move on to Java client.

Authenticating Java client

Java client has far fewer configuration options than either the .NET or JavaScript client. The only option available for authentication is `withAccessTokenProvider` call, which can be added to `HubConnectionBuilder.create` call when SignalR connection object is created. The function placed inside `withAccessTokenProvider` call will return a string, which will be applied as the bearer token.

And this concludes the overview of applying authentication on all of our clients. We will now launch our applications and see how they work.

Retrieving access token from the SSO provider

If you haven't got `AuthProvider` application running, launch it by executing `dotnet run` inside of its project folder. Then, launch `SignalRServer` application.

When you open the homepage of the `SignalRServer` application in the browser, you will be created with the login screen that looks like this:

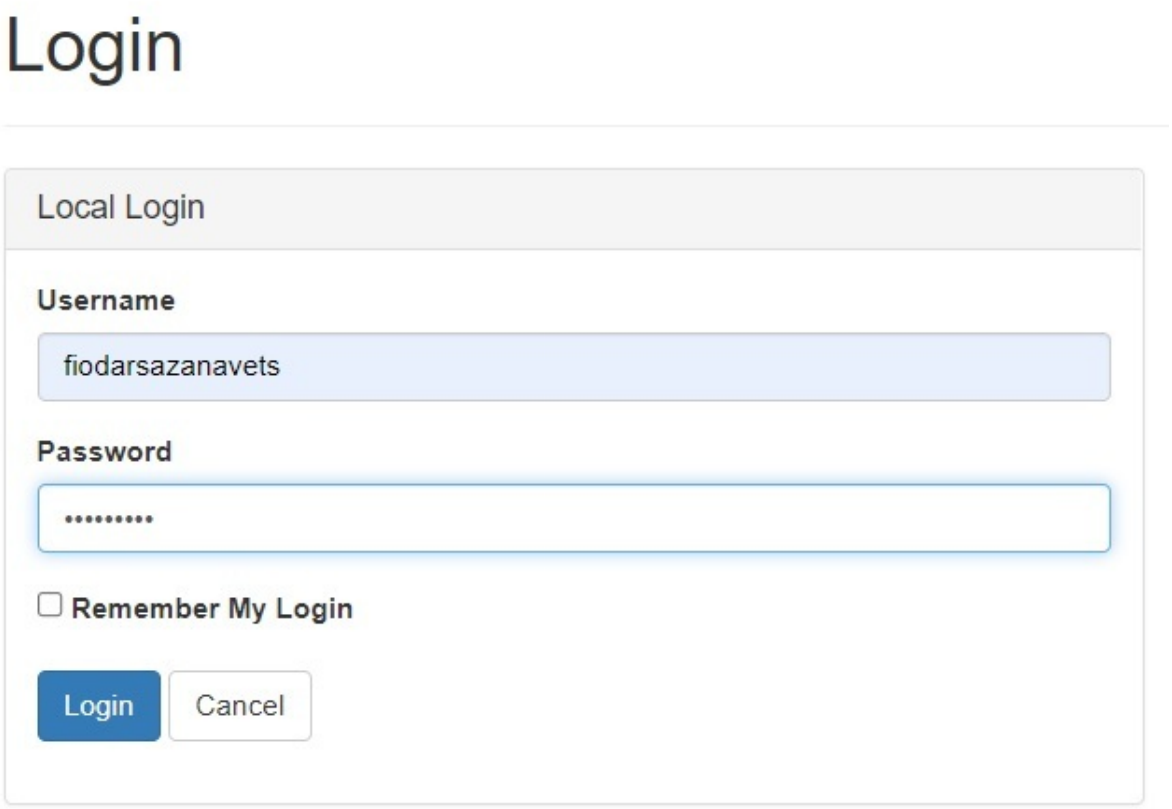

Figure 8.7 - Login screeen redirection

After a successful login by using one of the user credentials you've created earlier, you get redirected back to the homepage. And the authentication token will be printed in the application console:

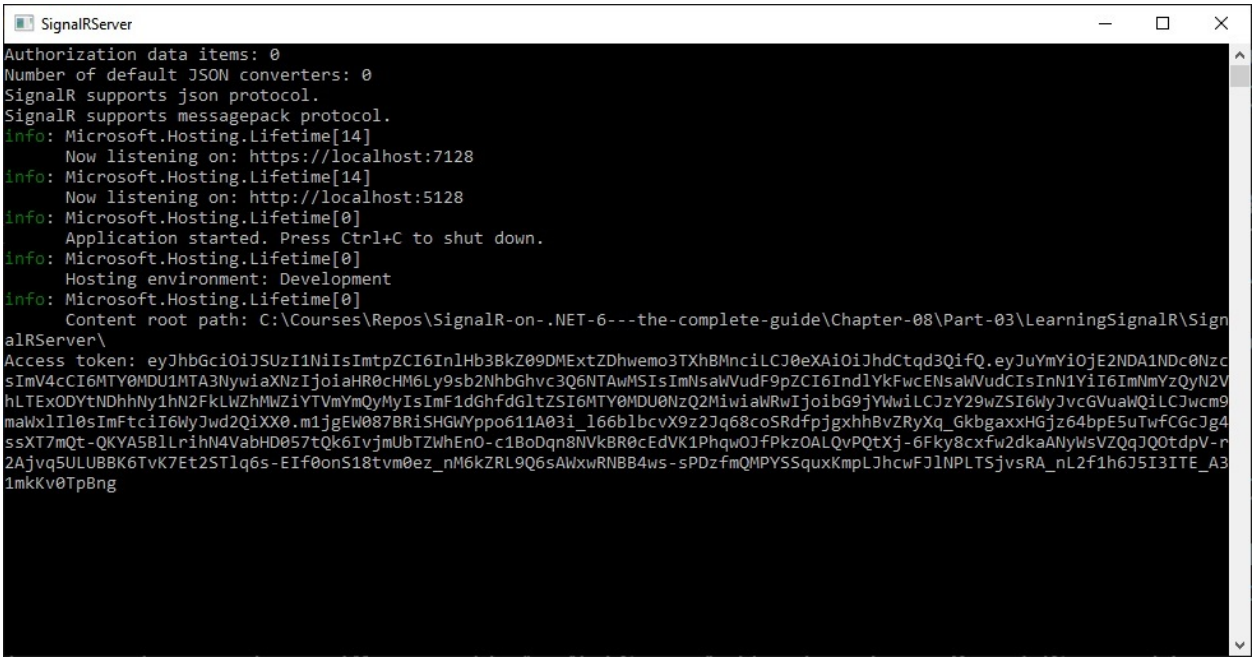

Figure 8.8 - Authentication token printed out in the console

Let's now have a closer look at the structure of the JWT.

JWT format structure

JWT consists of three parts, each of which is a base64-encoded string. Inside the token, the parts are separated by dots. And each of them can be read by decoding the string.

The first part is the header. It provides information on what type of token it is and what hashing algorithm it uses. Typically, it would look like this when decoded:

```
{
    "alg": "HS256",
    "typ": "JWT"
}
```

The second part is the payload. It is the actual object that contains the claims. This is an oversimplified example of the payload structure:

```
{
    "sub": "1234567890",
    "name": "John Smith",
    "iat": 1516239022
}
```

The third part is the signature, which only works when secret is used. Without the signature, anyone could forge a token. But because it uses the same secret that is registered in both SSO provider and the client application, it provides a reliable way of determining that the token came from a trusted source. The formula used to generate the signature is as follows:

```
HMAC_SHA256( secret, base64urlEncoding(header) + '.' + base64urlEncoding(payload))
```

And now we will have a look at the structure of our real token. We can do so by visiting `https://jwt.io` website and pasting the token from the console of `SignalRServer` application. The token will be automatically decoded in the browser and you will see the result similar to this:

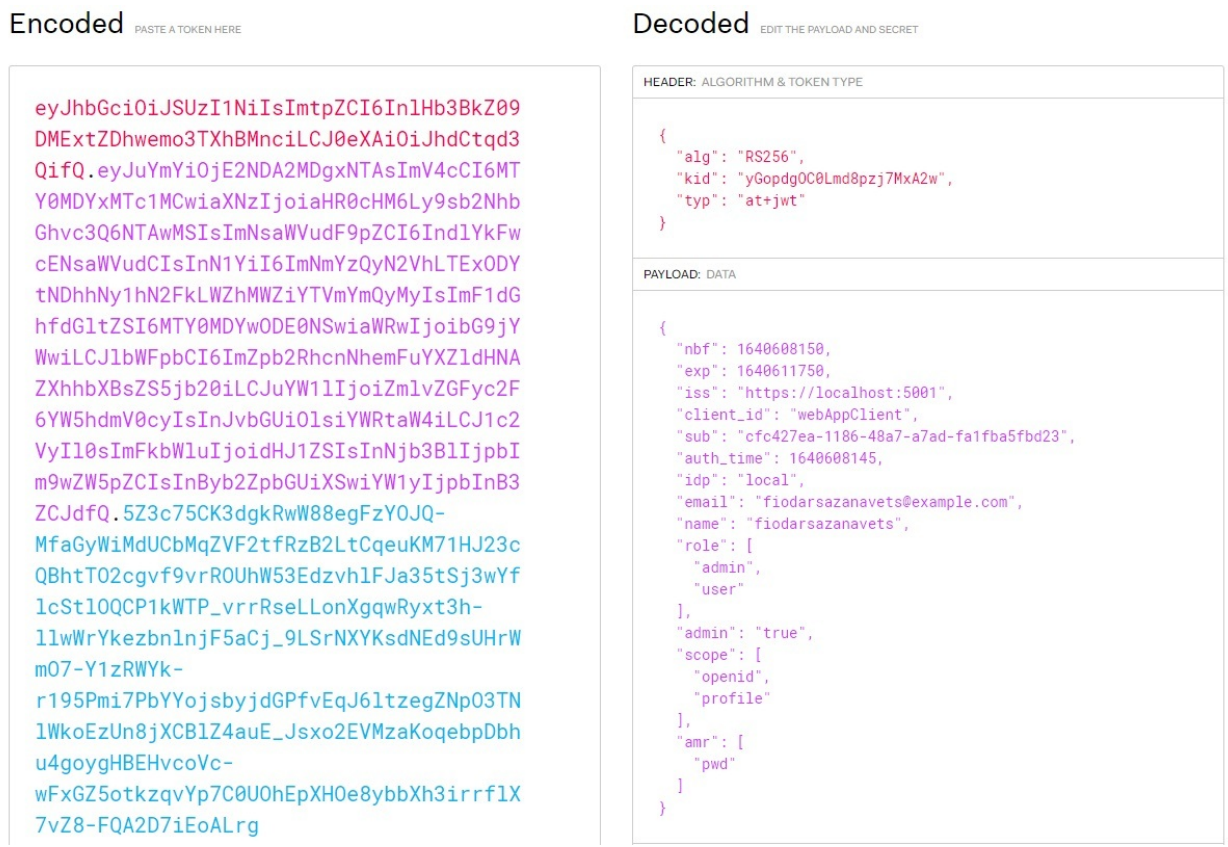

Figure 8.9 - JWT decoded

We can also launch our `DotnetClient` application and paste the token there when prompted. This will authenticate our client.

One thing to note though. Tokens have expiry time and normally applications have logic that would refresh the token at specific intervals. We didn't add such logic to our application, as the aim of this chapter is to only show enough of authentication concepts to make them applicable to SignalR. So, if our token expires, we can refresh it by logging out and logging back in.

And this concludes the overview of how to apply authentication to SignalR endpoints. Next, we will have a look at how to apply authorization.

Applying authorization in SignalR

Authentication on its own is good, but it's rarely sufficient. You don't only need to ensure that only known users can access your application, but that only those users that are permitted to use a specific resource can access it. And this is what the role of authorization is.

There are standard HTTP response codes that demonstrate the difference between authentication and authorization. 401 (Unauthorized) indicates that the user's credentials haven't been supplied or are invalid. 403 (Forbidden), one the other hand, is returned when the server is happy with the user's credentials, but the user doesn't have special privileges to access a specific resource.

There are several different types of authorization available in ASP.NET Core and all of them are applicable to SignalR. We just need to configure authorization handlers in our application. There are multiple ways of doing it and we will cover some of them.

Creating a custom requirement

One of the ways of applying authorization is to add a custom requirement class that inherits from `AuthorizationHandler` base class. We will create such a class. This will be done by creating `RoleRequirement.cs` file inside `SignalRServer` project folder. The content of the file will be as follows:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.SignalR;
using System.Security.Claims;

namespace SignalRServer
{
    public class RoleRequirement : AuthorizationHandler<RoleRequirement, HubInvocati\
onContext>,
    IAuthorizationRequirement
    {
        private readonly string requiredRole;

        public RoleRequirement(string requiredRole)
        {
            this.requiredRole = requiredRole;
        }

        protected override Task HandleRequirementAsync(AuthorizationHandlerContext c\
ontext,
            RoleRequirement requirement,
            HubInvocationContext resource)
```

```
        {
            var roles = ((ClaimsIdentity)context.User.Identity).Claims
                .Where(c => c.Type == ClaimTypes.Role)
                .Select(c => c.Value);

            if (roles.Contains(requiredRole))
                context.Succeed(requirement);

            return Task.CompletedTask;
        }
    }
}
```

When this class is registered in authorization middleware, `HandleRequirementAsync` method will be called when we receive a request, from which we have already extracted authentication token. In this specific instance, we are simply checking whether a role that we specified exists inside the claims.

One of the types that we are using in this class is `HubInvocationContext`. We aren't using it in our flow, but this is the class that tells us that we expect to run this requirement in the context of a request to a SignalR hub. But it's not there merely as a marker. We can actually apply some logic on the `resource` parameter of this type.

When we call `context.Succeed`, we tell the middleware that the requirement has been met. Otherwise, the request wouldn't be authorized.

Now, we will configure our application to actually use this requirement in its authorization middleware.

Configuring authorization middleware in ASP.NET Core application

We will first need to add the following namespace reference to the `Program.cs` file of `SignalRServer` project:

```
using SignalRServer;
```

Then, we will add the following code anywhere before `builder.Build` call:

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("BasicAuth", policy =>
    {
        policy.RequireAuthenticatedUser();
    });

    options.AddPolicy("AdminClaim", policy =>
    {
        policy.RequireClaim("admin");
    });

    options.AddPolicy("AdminOnly", policy =>
    {
        policy.Requirements.Add(new RoleRequirement("admin"));
    });
});
```

This code demonstrates different ways of how we can add the so-called authorization policies. The first policy we have added has been given the name of `BasicAuth`. This policy merely requires that only authenticated users can access the endpoint that it's applied on. This makes its behavior pretty much the same as using `Authorize` attribute on its own without specifying any additional parameters.

The next policy, `AdminClaim`, requires that the token contains the claim called `admin`. It doesn't matter what specific values that claim contains. The authorization will be successful if this claim is present at all. However, `RequireClaim` method can also accept an array of allowed values as the second parameter. If supplied, the authorization would be successful only if the claim contains one of these values. This specific claim, `admin`, is the custom claim that we previously have given to the users that have been assigned an admin role.

Finally, `AdminOnly` policy is the policy where we apply the custom requirement object that we have created earlier. We simply add our requirement to the list of requirements on the `policy` object. And the role that this requirement is applied to is `admin`.

The configuration of our authorization middleware is now complete and we have a variety of rules in it. We can now apply specific rules to our SignalR hub endpoints.

Applying authorization rules to individual endpoints

We will start by modifying the top-level attribute that we have above `LearningHub` class definition. We will add `Policy` parameter to it, so it will look like the following:

```
[Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme + "," +
CookieAuthenticationDefaults.AuthenticationScheme, Policy = "BasicAuth")]
```

This code demonstrates how an authorization policy can be applied to an endpoint in ASP.NET Core, whether it's in a SignalR hub, a controller, or a gRPC service. Behavior-wise, however, we haven't made any change, because `BasicAuth` policy merely ensures that authenticated users can use the resource. But the `Authrorize` attribute did it already.

Next, we will add the following attribute above `SendToGroup` hub method:

```
[Authorize(Roles = "user")]
```

We are not applying any named policies here. We are telling the endpoint that it can be only accessed by those users that have `user` role assigned. The `Roles` parameter accepts a comma-separated list of roles. And it will only be accessible if one of those roles exists inside the `role` claim of the authentication token payload.

Finally, we will now have a look at another way of applying a named policy inside an `Authorize` attribute. We can place the following attribute above the `AddUserToGroup` method:

```
[Authorize("AdminOnly")]
```

Policy name is the default constructor parameter in `Authorize` attribute. So, if this is the only parameter that you need to pass, you don't have to specify the parameter name explicitly.

You can now try all of these endpoints and see how they behave when you try to access them by using different user logins with different roles assigned. And this concludes the chapter on securing SignalR endpoints. Let's now summarize what we've learned.

Summary

In this chapter, you have learned that CORS stands for Cross-Origin Resource Sharing. CORS configuration is used to allow or disallow clients hosted on specific domains to access your application via HTTP.

You have also learned that single sign-on (SSO) is a system where everything related to authentication and authorization is managed by a specialized application. Other applications connect to this authentication provider when a user login is required. During a successful login, an encoded JSON token is generated, which contains the information about the user.

You now know that authentication is the process that confirms that the users are who they say they are. In order to be authenticated, a user needs to supply a verifiable information that is, in theory, is only known to them and to the server. This may include a password that matches the username, the client certificate of the user, one-time access code, etc.

You also now know the difference between authentication and authorization. While the former exists to confirm that the user is who they say they are, the latter confirms whether or not the user is allowed to access any specific resources. For example, the user may be registered on the system, but might not be assigned to a specific role that the endpoint requires.

In the next chapter, we will have a look at how to scale SignalR applications while maintaining the ability to message specific clients.

Test yourself

1. How do you prevent clients from different domains from connecting to your SignalR Hub?
   A. Firewall rules
   B. CORS configuration
   C. SSL
   D. TLS
2. What is single sign-on?
   A. Authentication system provided by Google
   B. A component of Windows Active Directory
   C. A passwordless system
   D. A system where authentication is shared between multiple applications
3. What is the difference between authentication and authorization?
   A. These terms are interchangeable
   B. Authentication doesn't use passwords, while authorization does
   C. Authentication tells you who the user is, while authorization tells you whether the user has sufficient privileges to perform a particular action
   D. Authentication doesn't require SSL, while authorization does

Further reading

CORS configuration on ASP.NET Core: https://docs.microsoft.com/en-us/aspnet/core/security/cors

Security considerations in ASP.NET Core SignalR: https://docs.microsoft.com/en-us/aspnet/core/signalr/security

Overview of ASP.NET Core authentication: https://docs.microsoft.com/en-us/aspnet/core/security/authentication

Introduction to ASP.NET Core Identity: https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity

Authentication and authorization in ASP.NET Core SignalR: https://docs.microsoft.com/en-us/aspnet/core/signalr/authn-and-authz

Official OpenID Connect documentation: https://openid.net/connect/

OAuth 2.0 documentation: https://oauth.net/2/

IdentityServer 4 documentation: https://identityserver4.readthedocs.io/en/latest/

Using cookie authentication without ASP.NET Core Identity: https://docs.microsoft.com/en-us/aspnet/core/security/authentication/cookie

9 - Scaling out SignalR application

You can run a single monolithic instance of a server application only if the number of clients accessing your application doesn't exceed a couple of thousand. But what if you expect hundreds of thousands, or even millions, of clients to connect to your application simultaneously? Then a single instance of an application won't be able to handle the load and you will need to scale the application out.

Scaling out a stateless application is relatively easy. If your web application merely accepts HTTP requests and then sends a query to a different application or a service that stores the data, you can have as many instances of such a web application as you want. To the client, it doesn't make any difference which specific instance it connects to. The outcome will be exactly the same.

But what would you do if your application needs to maintain a persistent connection with each client, like it is the case with SignalR? After all, you would still need to maintain the ability to send messages to specific clients and groups. But what if those groups are connected to a different instance of the application?

Fortunately, SignalR comes with a scaling-out mechanism that will allow multiple instances of your application act as if it's a single application. And this is what we will be talking about in this chapter.

The chapter consists of the following topics:

- Setting up Redis backplane
- Running multiple hub instances via Redis backplane
- Using HubContext to send messages from outside SignalR hub

By the end of this chapter, you will have learned how to use Redis to build a distributed SignalR hub that can support as many client connections as needed.

Prerequisites

This chapter assumes that you already have set up your development environment, as described in **chapter 1**. You will need the following:

- A machine with either Windows, Mac OS or Linux operating system
- A suitable IDE or code editor (Visual Studio, JetBrains Rider or VS Code)
- .NET 6 SDK (or newer)

Also, since we are continuing to build on top of the application that we have developed in the previous chapter, we need the code that we have written previously. If you have skipped the previous chapter, you can access the complete code from the following location in the GitHub repository:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide/tree/main/Chapter-08/Part-04/LearningSignalR

The complete code samples from this chapter are available from the following location in the GitHub repo, which has separate folders corresponding to individual parts of the chapter:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide/tree/main/Chapter-09

Setting up Redis backplane

There are two primary ways of scaling out SignalR hub - Azure SignalR Service and Redis backplane. Azure SignalR Service is what we will cover in the next chapter. And it's not suitable for all scenarios, as you will require Azure subscription and your hub will be hosted in Microsoft cloud. Redis backplane, on the other hand, is what allows you to host your SignalR hub anywhere, even on premises.

Redis backplane allows the hubs to exchange information between the instances of SignalR hubs connected to it. So, when you have your client connected to one instance of the hub and another client that you want to send a message to is connected to another hub instance, Redis backplane will transfer the message to the right instance of the application, so it can then be sent to the right client.

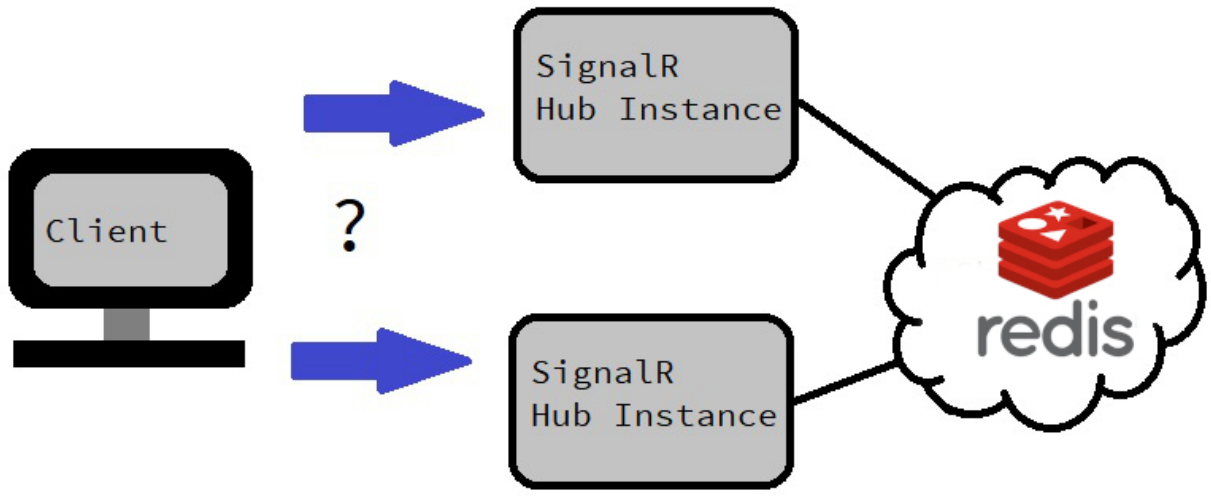

Figure 9.1 - Distributed SignalR application overview

To set up Redis backplane, we will need to first set up Redis service. Redis is an open-source (BSD licensed) in-memory data store system. It is primarily used as in-memory cache, but it can also be used for other purposes, such as pub-sub system.

To set up a SignalR backplane, you don't need to know how Redis works. You just need to have a Redis server running, so you can pass its URL into SignalR configuration. And you can install and run Redis on your development machine, no matter which operating system you use.

Running Redis on Linux

Installation instruction for Linux operating is available on the official website and is accessible via this link:

https://redis.io/download

You would just need to choose to either download the source code and compile Redis yourself, or use one of package installers, such as `apt-get` or `snapcraft`. Different ways of installing Redis are provided in the article supplied in via the link above. And so is the launch instruction.

Once you install and launch Redis, by default, it will be hosted on `localhost` address (IP `127.0.0.1`) and port 6379. But you can change this configuration if you need to.

Running Redis on Mac

As Mac is a Unix-based OS, just like Linux, Redis will work on it too. You can either download the source code from its official page and compile it yourself, as per instruction on its official download page[15], or you can install Redis via Homebrew package manager.

To install Homebrew, you can follow the instructions on the following webpage:

https://brew.sh/

And then, once it's installed, you can run the following command to install Redis:

```
brew services start redis
```

If you install Redis via Homebrew, it will be running in the background automatically. Otherwise, if you have compiled it yourself, you would need to launch it by executing the following command:

```
redis-server
```

As on Linux, default, Redis will be accessible via the localhost address and port 6379.

Running Redis on Windows

Because Redis is primarily designed to run on Linux, you will need to set up Windows Subsystem for Linux (WSL) on your Windows machine to run it. To enable WSL, you will need to open PowerShell and execute the following command:

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux
```

After you've enabled WSL on your machine, you will need to reboot the machine.

You will then need to download and install one of the supported Linux distros from Windows store. The supported distros are Ubuntu and Debian. You will just need to get the latest version.

Once installed, you can install Redis by executing the following commands:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install redis-server
redis-cli -v
```

To make sure that the Redis server is running, you can execute the restart command:

[15] https://redis.io/download

```
sudo service redis-server restart
```

This will restart Redis background service if it was running already and will start it if it wasn't running.

Redis server comes with a handy tool called `redis-cli` that you can use to make sure that the server is running. To access this tool, you will need to execute the following command:

```
redis-cli
```

Then, you can execute an insertion command. For example, you can set `client` record with the id of 1 and the value of `Test` by executing the following:

```
set client:1 "Test"
```

Then, to verify that Redis cache is working as it should, you can run the following command to obtain the value from `client` record with id of 1:

```
get client:1
```

And, if everything works correctly, you should see `Test` as the output.

Running Redis on Docker (all operating systems)

Docker is a containerization platform. It allows your services to run in isolated consistent mini-environments, known as containers. Many common software applications are available as Docker containers, so you don't have to install them on your actual machine.

You don't have to use Docker, but it would be useful to know it as another way of launching your Redis on any operating system.

To download the latest version of Docker, you can follow this link:

https://docs.docker.com/get-docker/

Then, once set up, you can pull the latest Redis container image from the default registry. The following page contains the setup instruction:

https://hub.docker.com/_/redis

Because your Redis container will have its port mapped to a port on your host machine, you would still be able to access it via localhost URL.

Once our Redis server is running, we can implement it inside our ASP.NET Core application that hosts the SignalR hub.

Running multiple hub instances via Redis backplane

Normally, when your application is scaled out, you will be running multiple instances of the same application that was created from the same code. And there are different systems that allow you to scale applications: Docker Swarm, Kubernetes, Azure Service Fabric, etc.

The entry point into the application for the clients would typically be a reverse proxy, a load balancer or a combination of both. Basically, some gateway software would accept the initial request. And then this request would be re-routed to a specific instance of the application. Typically, there would be some algorithm in place to ensure that it's an instance with a reasonably low load that the request gets routed to.

There are many different ways you can scale your application. But the scope of this chapter is to show you general principles of working with SignalR backplane rather than showing you a specific way of scaling out your web application. This is why we won't cover any specific scaling-out techniques. Instead, we will just manually create two copies of a web application and connect them to the same instance of the Redis backplane. Then, we will just connect different clients directly to different web application instances and verify that they can communicate with each other via SignalR.

Setting up multiple SignalR hubs

To connect your SignalR hub to Redis backplane, you will first need to install `Microsoft.AspNetCore.SignalR.StackExchangeRedis` NuGet package in the application that hosts the hub. To install it, just run this command inside the project folder of that application, which, in our case, is `SignalRServer`:

```
dotnet add package Microsoft.AspNetCore.SignalR.StackExchangeRedis
```

Next, we will locate the call to `builder.Services.AddSignalR()` and replace it with the. following:

```
builder.Services.AddSignalR().AddStackExchangeRedis("< your Redis connection string>\
");
```

You can use any standard Redis connection string format. If you have Redis running locally on the same machine that you are hosting SignalR hub on and it runs on the default port, the connection string would be `localhost:6379` or `127.0.0.1:6379`.

It's up to you if you want to replace the call to `AddSignalR` or just append `AddStackExchangeRedis` at the end of it. But if you want to remove the MessagePack configuration that we have added to it in **chapter 7**, you will also need to remove calls related to MessagePack protocol from the clients that you still intend to use. Otherwise, the clients will be sending MessagePack messages and the server won't know how to read them.

That's all we needed to do to make the web application that hosts SignalR hub scalable. Now, we could have just copied the content of `SignalRServer` project folder (or its compile outputs) into a different folder and changed ports in the URLs of `launchSettings.json` file. This would allow us to then launch two instances of an identical web application with different access points. But we will do something else instead.

Hosting the same SignalR hub in different applications

SignalR hub is structured in such a way that you can scale the hub independently from scaling the web application that hosts it. How is it possible, you may ask, if the hub is an integral part of the web application? Well, if you are using Redis backplane, you can have the same hub code in multiple different applications. And all these hub instances will still work as the same hub, even though the applications that host them are different.

There are some valid use cases for this approach. For example, you may want to have an MVC or Razor Pages web application with in-browser SignalR client. This way, if the SignalR hub was hosted by a separate service, the client-side configuration would be more complicated. And you would need to apply different configuration for different environment. However, if SignalR hub is hosted in the same web application that serves the page with the SignalR client to the browser, all you need is a relative path, which would be identical for all environments.

Perhaps you would need another service inside your distributed app that would use REST API or gRPC interface alongside your hub. Again, if some of your clients use SignalR and one of those other protocols, you can simplify their configuration by just using a single URL. And to do so, you would need to host SignalR hub inside the same application as your other API.

For example, you may have a service that IoT devices talk to. That service would have gRPC interface for the devices to make occasional client-initiated call and SignalR connection for the server application to issue real-time instructions to these devices. But what if you also need a web page that the users will be able to access to view real-time device status information, which is also managed by SignalR?

You could have it all in one application, but this will only work until you'll need to scale it. And because it's quite likely that the number of IoT devices that will be connected to the application will be different from the number of users looking at the devices, it would be more efficient to scale those two parts of the application separately. Therefore, it makes sense to have one application that serves web pages to the users and another one that has gRPC interface for the IoT devices to talk to.

But if you have yet another application that is purely dedicated to hosting SignalR hub, things will start getting excessively complicated. Firstly, there are above-mentioned issues with additional client configuration, where they will need to have additional endpoints added to it. Secondly, it would be harder to decide how to scale the separate SignalR service correctly. Thirdly, the topology of your distributed application would be harder to comprehend by someone who has never worked on it before. And this is why it's just better to have the same hub hosted by both the user-facing web app and the web API service that the IoT devices use.

The best way to share the hub between different host applications is to have it inside a class library that any application can reference. And this is what we are going to do here.

Moving SignalR hub to a class library

Inside our `LearningSignalR` solution folder, we will create a class library project by executing the following command:

```
dotnet new classlib -o SignalRHubs
```

Once the project has been created, we will add it to the solution by executing the following command:

```
dotnet sln add SignalRHubs\SignalRHubs.csproj
```

Next, we will open the terminal inside `SignalRHubs` project folder and will install the required NuGet packages by executing the following commands:

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
dotnet add package Microsoft.AspNetCore.SignalR.StackExchangeRedis
```

You can now also remove these package references from `SignalRServer.csproj` file, as this project will still obtain them by referencing `SignalRHubs` class library.

We will need the first package because our SignalR hub needs it, as it uses `Authorize` attributes associated with JWT. And we will need the second package so we don't have to install it on individual project that will use this class library. All of them will then be able to connect to Redis backplane.

Next, we will move `LearningHub.cs` and `ILearningHubClient.cs` files from `SignalRServer` project to `SignalRHubs`. In each one of those, we will need to replace the existing namespace with `SignalRHubs`.

Our code will no longer compile, so we will need to make some changes to it. We will first add `SignalRHubs` project reference to `SignalRServer` project. You can either do it via an IDE, or you can manually add the following snippet to `SignalRServer.csproj` file:

```
<ProjectReference Include="..\SignalRHubs\SignalRHubs.csproj" />
```

This can go inside any `ItemGroup` element, but to make your file structure clean, it would make sense to place it alongside the existing project references.

Next, we will open `Program.cs` file. In there, we will locate the `SignalRServer.Hubs` namespace reference and we will replace it with the following:

```
using SignalRHubs;
```

Now, our code will compile again. What we will need to do next is add another application that will be hosting the same hub.

Adding another web application to host the hub

We can add absolutely any ASP.NET Core app to our application, as SignalR hub would be compatible with any of them. And the setup would be almost identical, regardless of whether you have chosen MVC, Web API or just an empty ASP.NET Core template.

For example, to create another MVC application, you would need to open your terminal inside the `LearningSignalR` solution folder and execute the following command:

```
dotnet new mct -o SignalRServer2
```

We can then add this application to the solution by executing the following command:

```
dotnel sln add SignalRServer2\SignalRServer2.csproj
```

Then, we will add the following markup to the `SignalRServer2.csproj` file to ensure that it references `SignalRHubs` class library:

```
<ItemGroup>
    <ProjectReference Include="..\SignalRHubs\SignalRHubs.csproj" />
</ItemGroup>
```

Next, we will open `Program.cs` file inside `SignalRServer2` project and will add the following namespace reference to it:

```
using SignalRHubs;
```

Afterwards, we will add the following line anywhere before the `Build` call on `builder` object:

```
builder.Services.AddSignalR().AddStackExchangeRedis("< your Redis connection string>\
");
```

The connection string needs to be the same as the one we have defined in `SignalRServer` project.

And finally, we will register the hub endpoint by adding the following line just before `Run` call on `app` object:

```
app.MapHub<LearningHub>("/learningHub");
```

Then, there is either of two things you can do. You could either copy all authentication and authorization setup calls from the `Program.cs` file of the `SignalRServer` project to `Program.cs` file of `SignalRServer2` project. Or you could simply remove all `Authorize` attributes from `LearningHub` class of `SignalRHubs` project. Either option is fine.

Now you have ended up with two different ASP.NET Core applications that host exactly the same version of SignalR hub and use exactly the same Redis instance to connect those hubs together. Both of the applications would have been generated with different ports for the default URL, so you can launch them both at the same time. Let's do it and see how Redis backplane operates.

Launching distributed SignalR hub

We will now launch both `SignalRServer` and `SignalRServer2` applications. You can either do it from the IDE, or you can execute `dotnet run` command from inside of each of the project folders.

Once the applications are up, you can navigate to the main page of the `SignalRServer` application. Then, you can launch `DotnetClient` application and, when prompted, get it to connect to the SignalR hub of the `SignalRServer2` application. This would be the base URL of the application, as defined in `launchSettings.json` file inside the project, followed by `/learningHub` path.

Now, to verify that everything has been set up correctly, you can broadcast a message from `DotnetClient` app to all clients (either option `0` or `1`). And even though the message was sent to the hub on `SignalRServer2` app, it should appear on the open homepage of `SignalRServer` app, because both of the hubs are connected by the same Redis backplane. And this is how distributed SignalR hub works.

But there is more to it. What if we wanted to send a message to any of the hub clients from outside the actual hub? For example, we may have some backround process running inside of our web application, which occasionally sends some updates to the clients. But we can't make direct calls on the methods from our SignalR hub, as those are only meant to be accessed by the clients.

Fortunately, there is a way to do it. And this is what we will have a look at next.

Using HubContext to send messages from outside SignalR hub

SignalR comes with `IHubContext` interface, the implementations of which allow you to access all fields of a specific SignalR hub, including its groups, clients and so on. You don't have to worry about registering any concrete implementations of this interface yourself. If you inject this interface into the constructor of any class, an appropriate implementation will be automatically resolved. All the dependencies get automatically registered when you call `AddSignalR` method on `Services` field of `builder` object inside `Program.cs` class.

`HubContext` is not designed specifically for a scaled SignalR hub. You can use it with monolithic SignalR hubs too. But because it's especially useful in the context of a distributed SignalR hub, we will talk about it in this section.

Implementing HubContext in our application

To demonstrate how `HubContext` works, will now open `HomeController.cs` file, which is located in `Controllers` folder of `SignalRServer` project. First, we will make sure that the file has the following namespace references (which, on .NET 6, you could alternatively make global if they are already referenced elsewhere):

```
using Microsoft.AspNetCore.SignalR;
using SignalRHubs;
```

Then, there are two ways we can create a `HubContext` field. You can just specify the name of the hub, like this:

```
private readonly IHubContext<LearningHub> hubContext;
```

Or you can use strongly-typed hub context, which will be defined like this:

```
private readonly IHubContext<LearningHub, ILearningHubClient> hubContext;
```

Then, whichever type of `HubContext` we've chosen, we need to inject it into the constructor. So, it will be done like this for a standard `HubContext`:

```
public HomeController(IHubContext<LearningHub> hubContext)
{
    this.hubContext = hubContext;
}
```

Or it will be done like this for a strongly-typed `HubContext`:

```
public HomeController(IHubContext<LearningHub, ILearningHubClient> hubContext)
{
    this.hubContext = hubContext;
}
```

Next, we will add the following line before the return statement of the `Index` action method:

```
await hubContext.Clients.All.SendAsync("ReceiveMessage", "Index page has been opened\
 by a client.");
```

For a strongly-typed variety, use this line instead:

```
await hubContext.Clients.All.ReceiveMessage("Index page has been opened by a client.\
");
```

And now we can test our `HubContext` implementation and see how it works with a distributed hub.

Testing HubContext on a distributed SignalR hub

We will launch `SignalRServer` and `SignalRServer2` applications. Then, once these applications are up, we will launch `DotnetClient` application and connect it to the SignalR hub of `SignalRServer2` application.

Now, if you refresh the homepage of `SignalRServer` application, you should see the following message in the console of `DotnetClient` app:

```
Index page has been opened by a client.
```

Other than that, `HubContext` implementation would have all the same properties as `Hub` base class. Anything you can do from the hub implementation can be done from `IHubContext` implementation.

And this concludes an overview of how to scale SignalR hubs. Let's summarize what we have learned.

Summary

In this chapter, you have learned how to scale SignalR hub by using Redis backplane. You have also learned how SignalR hub can be scaled independently of scaling individual web applications. By using Redis backplane, multiple application can use the same SignalR hub definition and share messages across all hub implementations, even if those applications aren't just copies of each other.

You have also learned that there is a way to send messages to SignalR hub clients from outside the hub. To do so, you can use injectable `IHubContext` interface with the definition of the hub you want to use. It will be resolved automatically if you have registered SignalR dependencies.

`IHubContext` can use either a standard or a strongly-typed hub implementations. For the latter, you need to pass specific client interface into `IHubContext`. And you will then have to use actual client methods to call clients instead of the generic `SendAsync` method.

You have learned that `HubContext` is especially appropriate for the use inside of a scalable SignalR application. However, you can still use it inside a singular monolithic application.

In the next chapter, we will have a look at an alternative way of scaling out SignalR hub. You will learn how to do it by using Azure SignalR Service.

Test yourself

1. What is scaling out?
   A. Limiting the number of clients that are allowed to connect
   B. Running multiple copies of a particular server resource, so the load is evenly distributed between connected clients
   C. Using more powerful hardware to host the web application on
   D. Same as using Redis cache
2. What is SignalR backplane?
   A. A mechanism that allows multiple instances of a Hub to communicate with each other and act as a single Hub
   B. Low-level implementation details of SignalR
   C. Database that SignalR uses
   D. Cache that SignalR uses for the storage of connection ids and groups
3. What is HubContext?
   A. HTTP Context under which the logic inside of any given Hub is executed
   B. The list of all connected Hub clients
   C. An object that is positioned outside any SignalR Hub that can instruct the Hub to send messages to particular clients
   D. A particular client connection to a SignalR Hub

Further reading

SignalR Redis backplane documentation: https://docs.microsoft.com/en-us/aspnet/core/signalr/redis-backplane

Official HubContext documentation: https://docs.microsoft.com/en-us/aspnet/core/signalr/hubcontext

Getting started with Redis: https://redis.io/topics/introduction

Docker tutorial: https://docs.docker.com/get-started/

Getting started with Kubernetes: https://kubernetes.io/docs/tutorials/kubernetes-basics/

Load balancing explained: https://www.nginx.com/resources/glossary/load-balancing/

10 - Introducing Azure SignalR Service

In the previous chapter, we had a look at how to scale a SignalR hub by using Redis backplane. But you can also scale your hubs by using Azure SignalR Service. And this is what we will have a look at in this chapter.

Using Azure SignalR Service has some advantages over using Redis backplane. One of its core advantages is that you will no longer have to host and scale SignalR hub yourself. All you have to do is set up the service on Azure by completing a relatively simple form. And then you can connect your existing application to this service by making very few code changes.

Perhaps the only disadvantages of using Azure SignalR Service is that you will be limited to be using Azure as your hosting provider. It may be an issue if the rest of your assets are hosted on AWS or Google Cloud. And, just like it is with any cloud-hosted service, you will also be charged for the usage. However, if your application is in the need of being scaled out, then chances are that you will have little choice other than hosting it in cloud anyway.

Perhaps one notable difference between Azure SignalR Service and Redis backplane is that the former is a complete method of scaling SignalR hub, while the latter is the way of making multiple hub instances communicate with each other while your application has already been scaled out by replication. When you use Azure SignalR Service, hub connection from a client will be re-routed directly to it, while with Redis backplane, you still need to connect to one of your own application instances directly. So, if SignalR hub is the only bottleneck in your application that needs to be scaled out, you don't even have to scale out your main application if you are using Azure SignalR Service. With Redis backplane, on the other hand, you still have to replicate the main ASP.NET Core application.

The chapter consists of the following topics:

- Setting up Azure SignalR Service
- Adding Azure SignalR Service dependencies to your application
- Overview of Azure SignalR Service REST API

By the end of this chapter, you will have learned how to set up Azure SignalR Service and how to connect your ASP.NET Core application to it.

Prerequisites

This chapter assumes that you already have set up your development environment, as described in **chapter 1**. You will need the following:

- A machine with either Windows, Mac OS or Linux operating system
- A suitable IDE or code editor (Visual Studio, JetBrains Rider or VS Code)
- .NET 6 SDK (or newer)

Also, since we are continuing to build on top of the application that we have developed in the previous chapter, we need the code that we have written previously. If you have skipped the previous chapter, you can access the complete code from the following location in the GitHub repository:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide/tree/main/Chapter-09/Part-03/LearningSignalR

The complete code samples from this chapter are available from the following location in the GitHub repo, which has separate folders corresponding to individual parts of the chapter:

https://github.com/fiodarsazanavets/SignalR-on-.NET-6---the-complete-guide/tree/main/Chapter-10

Setting up Azure SignalR Service

To create an instance of Azure SignalR Service, you will need to visit Azure Portal, which is accessible via the following URL:

https://portal.azure.com

In order to use this portal, you need to create a Microsoft account. If you don't have one already, you will be automatically prompted to create it. And the process of creating an account is straight-forward, as you will be guided through the process.

Once you have Microsoft account set up, you will be guided on how to register with Azure. Normally, if this is your first account, you will be given $200 worth of credit, so you can try various services on the platform.

Once you've registered on Azure, you can start creating services on it. And the process will be straight-forward, as all the forms on Azure Portal have been designed to be as informative and as user-friendly as possible.

What you need to do next is enter **Azure SignalR Service** in the search box. In the results, you will be presented with a service template of this type. You will need to click on it, and this will take you to the set-up form.

The setup form will be relatively simple to use. You will need to specify the name of the service, its host location, and some other set-up information, such as payment tier. It's up to you how you configure it, as all of the options will be explained to you on the form. But just bear in mind that it's best to choose the geographic location of the hosting datacenter as close to your potential users as possible (or to your own location if you only intend to use it for test purposes). This is to ensure that you have as little latency as possible.

Once the service is created, you can open it to obtain its connection string. And this is what we need for the next step of the setup process. We can start adding necessary code changes to make our application use the Azure SignalR Service instance we have just created.

Adding Azure SignalR Service dependencies to your application

To add Azure SignalR Service dependencies to our application, we need to execute the following command inside `SignalRHubs` project folder:

```
dotnet add package Microsoft.Azure.SignalR
```

Next, we can open `Program.cs` of `SignalRServer` project and replace the line with `AddSignalR` call with the following:

```
builder.Services.AddSignalR().AddAzureSignalR();
```

We can do the same in `SignalRServer2` project. And there are two ways of configuring it. `AddAzureSignalR` method can accept a string parameter representing the connection string of the Azure SignalR Service we have created earlier. But we can also leave it without the parameter. But to make it work, we will then need to use User Secrets tool inside the project.

If you don't want to pass the connection string in the code of your `Program.cs` file, you would need to execute the following command inside the project folder:

```
dotnet user-secrets init
```

Then, you would need to execute the following command:

```
dotnet user-secrets set Azure:SignalR:ConnectionString "<Azure SignalR Service conne\
ction string"
```

`Azure:SignalR:ConnectionString` is the key of User Secrets tool that will be looked up by the middleware if no connection string is explicitly specified.

That's all we need to do to connect our app to Azure SignalR Service. Your client connection will now be re-routed to it. However, the clients will still be able to trigger methods in your own Hub class. Only that anything that sends messages to the clients or groups thereof inside those methods will then reroute those calls to the cloud, so your own application won't have to deal with the load.

There is only one extra thing that we need to apply to it if our existing app was using `HubContext` outside the hub, as the default `IHubContext` implementation isn't compatible with Azure SignalR Service connection. But luckily, we won't have to apply many code changes to make it work, as you shall see shortly.

Making HubContext work with Azure SignalR Server

As you may recall from the previous chapter, `HubContext` is a mechanism that allows you to send messages to SignalR hub clients from outside the hub class. For example, it is useful if you have a background process that needs to update clients on regular basis. Likewise, you may have some event listener in your server application that updates clients whenever some event gets triggered.

If we use a monolithic SignalR hub or scale it our via Redis backplane, the implementation of `IHubContext` interface will be automatically injected into the constructor of a class that needs to use it. Calling `AddSignalR` method on the service collection will ensure that all dependencies will be applied and that all appropriate implementations will be automatically resolved. However, this will not work if you use Azure SignalR Service. But the good news is that there is a library that allows you to inject an implementation of `IHubContext` that is specific to Azure SignalR Service. So, you will still be able to inject `IHubContext` instances into the classes. You will just need to make some changes to dependency injection logic.

Before we start, we need to install a NuGet package that contains the `IHubContext` implementation that we need. To do so, we will need to execute the following command inside `SignalRHubs` project folder:

```
dotnet add package Microsoft.Azure.SignalR.Management
```

Then, we will open `Program.cs` file of our `SignalRServer` project and add the following namespace reference to it:

```
using Microsoft.Azure.SignalR.Management;
```

We will then add the following code just before `builder.Build` call, replacing `<Azure SignalR Service connection string>` with the actual connection string that you can obtain from Azure SignalR Service instance:

```
var serviceManager = new ServiceManagerBuilder()
    .WithOptions(option =>
    {
        option.ConnectionString = "<Azure SignalR Service connection string>";
    })
    .BuildServiceManager();

var hubContext = await serviceManager.CreateHubContextAsync<LearningHub>("LearningHu\
b", CancellationToken.None);

builder.Services.Add(new ServiceDescriptor(typeof(IHubContext<LearningHub>), hubCont\
ext));
```

In our previous chapter, we have injected an instance of `IHubContext<LearningHub>` into the constructor of `HomeController` class. So, we just need to create another implementation of the same type and inject it into the application to make sure that we overwrite the default implementation.

`ServiceManagerBuilder` class allows us to build a service manager object that are specific to the instance of Azure SignalR Service implementation that we have connected to via our connection string. Then, we can call `CreateHubContextAsync` method on the service manager object to create an implementation of `IHubContext` that will work with AzureSignalR. This method accepts the name of the hub and a cancellation token. If we want an implementation of `IHubContext` that is specific to a particular SignalR hub class, we heed to specify the name of the class in the angle brackets before the method parameters. In our case, if we specifically need an implementation of `IHubContext<LearningHub>`, we just need to specify `LearningHub` in the angle brackets.

Finally, we can add this instance of `IHubContext` implementation to our dependency injection container. We do so by calling `Add` on `Services` property of `builder` object. The parameter of this method is an instance of `ServiceDescriptor`. The first parameter that we pass into it is the name of the type that we are implementing, which, in our case, is `IHubContext<LearningHub>`. The second parameter is the actual implementation, which, in our case, is the instance of the class that we have created. And we do this call as the last step before the `app` object is built. This is to make sure that we overwrite any existing implementation.

And this is it. The existing code that used an implementation of `IHubContext` will still work with no changes. It will simply receive a different implementation of this interface.

But `HubContext` is not the only way you can send messages to a SignalR hub hosted inside Azure SignalR Service from your server application. Azure SignalR Service also has an inbuilt REST API, which allows you to send messages to it by making standard HTTP requests. And this is what we will have a look at next.

Overview of Azure SignalR Service REST API

The advantage of using REST API to send messages to Azure SignalR Service is that you won't need to install any additional NuGet packages. You can use any HTTP clients, including the inbuilt ones. The disadvantage of using REST API is that you will have to write additional code that will convert the objects that are natively used by the hub into JSON payload of HTTP requests. And you will also need to implement your own authentication logic.

An now we will go ahead and add some code for sending messages to the hub by using a standard `HttpClient` class. We will do so inside `HomeController.cs` file of `SignalRServer2` project. And before we start, we will need to ensure that we have all of the following namespace references:

```
using Newtonsoft.Json;
using System.Net.Http.Headers;
using System.Text;
```

Then, we will add the following private field and the constructor, replacing `<instance-name>` with the actual instance name from Azure SignalR Service instance:

```
private readonly string signalRHubUrl;

public HomeController()
{
    signalRHubUrl = "https://<instance-name>.service.signalr.net/api/v1/hubs/learnin\
gHub";
}
```

Then, we can add the following code to the `Index` action method, where we replace `<your JWT>` with the actual JWT value we can obtain from Azure:

```
using var client = new HttpClient();

var payloadMessage = new
{
    Target = "ReceiveMessage",
    Arguments = new[]
    {
        "Client connected to a secondary web application"
    }
};

var request = new HttpRequestMessage(HttpMethod.Post, new UriBuilder(signalRHubUrl).\
Uri);
request.Headers.Add("Authorization", "Bearer <your JWT>");
request.Headers.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));
request.Content = new StringContent(JsonConvert.SerializeObject(payloadMessage), Enc\
oding.UTF8, "application/json");

var response = await client.SendAsync(request, HttpCompletionOption.ResponseHeadersR\
ead);

if (!response.IsSuccessStatusCode)
    throw new Exception("Failure sending SignalR message.");
```

So, here is what we've done. Whenever somebody opens the home page of `SignalRServer2` application in the browser, all SignalR clients that are connected to the same hub hosted by the same Azure SignalR Service instance will receive the following message:

```
Client connected to a secondary web application
```

This is because, when we are sending the request, we are choosing the generic URL type that will broadcast the message to all clients. We will be triggering `ReceiveMessage` event listener on our clients, as this is what we have specified as our `target` field in the payload. The parameters of this event listener are specified by `arguments` field.

But this is not the only thing that you can do with the REST API of Azure SignalR Service. Here is the full list of the endpoints you can use.

Full list of Azure SignalR Service REST API endpoints

Broadcast a message to all clients connected to target hub

```
POST /api/v1/hubs/{hub}
```

Broadcast a message to all clients belong to the target user

```
POST /api/v1/hubs/{hub}/users/{id}
```

Send message to the specific connection

```
POST /api/v1/hubs/{hub}/connections/{connectionId}
```

Check if the connection with the given connectionId exists

```
GET /api/v1/hubs/{hub}/connections/{connectionId}
```

Close the client connection

```
DELETE /api/v1/hubs/{hub}/connections/{connectionId}
```

Broadcast a message to all clients within the target group

```
POST /api/v1/hubs/{hub}/groups/{group}
```

Check if there are any client connections inside the given group

```
GET /api/v1/hubs/{hub}/groups/{group}
```

Check if there are any client connections connected for the given user

```
GET /api/v1/hubs/{hub}/users/{user}
```

Add a connection to the target group

```
PUT /api/v1/hubs/{hub}/groups/{group}/connections/{connectionId}
```

Remove a connection from the target group

```
DELETE /api/v1/hubs/{hub}/groups/{group}/connections/{connectionId}
```

Check whether a user exists in the target group

```
GET /api/v1/hubs/{hub}/groups/{group}/users/{user}
```

Add a user to the target group

```
PUT /api/v1/hubs/{hub}/groups/{group}/users/{user}
```

Remove a user from the target group

```
DELETE /api/v1/hubs/{hub}/groups/{group}/users/{user}
```

Remove a user from all groups

```
DELETE /api/v1/hubs/{hub}/users/{user}/groups
```

As you can see, there are more actions you can do via the REST API than you could via `HubContext`. For example, you can send messages to all connections that are associated with a specific users. Azure SignalR Service supports this out of the box without having to implement a custom dictionary or associating user-specific connections with user-specific groups. In order to use this feature, all you need to do is add `nameid` attribute to the payload of your JWT. And this will be your user id that you can then specify in the URL.

Authenticating into Azure SirnalR Service REST API

The REST API of Azure SignalR Service uses standard JWT as bearer token inside `Authorization` header. Both OpenID Connect and OAuth protocols can be applied in the context of this REST API. The overview of these protocols is provided in the **chapter 8**. For specific application of these protocols in the context of Azure SignalR Service, you can visit **Azure SignalR Service authentication** link in the **further reading** section.

There are two important points to remember when applying JWT to Azure SignalR Service REST API endpoints:

1. The payload needs to have `aud` (audience) field and its value should match the URL that you are sending the request to.
2. The payload needs to have `exp` (expiry) field and it should contain epoch time of token's expiry.

And this concludes the chapter on Azure SignalR Service. Let's summarize what we've learned.

Summary

In this chapter, you have learned that you can scale your SignalR hub via Azure SignalR Service. To do so, you won't have to make any code changes to the existing hubs. You just need to add an additional NuGet package to your server application and apply additional method call to your middleware setup.

You have also learned that the standard `HubContext` doesn't work with Azure SignalR Service out of the box. If you use injectable `IHubContext` interface, you need to map it to a specific implementation from Azure SignalR Service Management NuGet package to make it compatible with Azure SignalR Service.

But now you also know that you don't have to use `HubContext` to send messages to Azure SignalR Service from your server. The service supports a standard REST API, which you can use without any additional NuGet packages.

And this concludes the book on applying SignalR library in the context of .NET 6.

Test yourself

1. What is Azure SignalR Service?
   A. Redis backplane hosted in Azure
   B. ASP.NET Core application hosted in Azure that has SignalR support
   C. A proxy used as a load balancer that is placed in front of SignalR applications hosted in Azure

   D. A service hosted in Azure that is used for scaling SignalR hub
2. Does Azure SignalR Service support HubContext
   A. Yes, it supports it and no changes are required
   B. No, it doesn't support it at all
   C. Yes, but only via a special NuGet package and additional middleware configuration
   D. Yes, but only via some additional middleware configuration
3. How can messages be sent to Azure SignalR Service from the server without installing any additional NuGet packages?
   A. You cannot do it without additional NuGet packages
   B. Via REST API
   C. Via gRPC interface
   D. Via WebSocket connection

Further reading

What is Azure SignalR Service?: https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-overview

Scale ASP.NET Core SignalR applications with Azure SignalR Service: https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-concept-scale-aspnet-core

Build real-time Apps with Azure Functions and Azure SignalR Service: https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-concept-azure-functions

REST API in Azure SignalR Service: https://github.com/Azure/azure-signalr/blob/dev/docs/rest-api.md

Azure SignalR Service REST API quickstart: https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-quickstart-rest-api

Azure SignalR Service Management SDK: https://github.com/Azure/azure-signalr/blob/dev/docs/management-sdk-guide.md

Safe storage of app secrets in development in ASP.NET Core: https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets

Azure SignalR Service authentication: https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-concept-authenticate-oauth

Wrapping up

This book has provided a complete guidance on how to use SignalR. After reading all the chapters, you should be fully equipped to apply the library in the context of your own problem domain.

But as any technology, SignalR evolves. Improvements to the library are expected in the future and it's quite likely that the information in this book will eventually become obsolete. This is why this book will be regularly updated as new editions.

But each new edition doesn't have to keep the same structure as this book with just a few version-specific code updates. Some additional topics may also be included.

Therefore, if there is anything that you would like to be covered in more depth in the next edition, this is how you can get in touch with me either via Twitter or LinkedIn. Likewise, feel free to get in touch if you found any errors in the current edition.

Twitter: https://twitter.com/FSazanavets

LinkedIn: https://www.linkedin.com/in/fiodar-sazanavets/

I hope that you found this book useful and I am looking forward to hearing from you if you have any questions of feedback.

Yours truly
Fiodar Sazanavets

Answers to self-assessment questions

Chapter 2

1. **C**. A class that contains the methods that clients can trigger
2. **D**. All of the above
3. **C**. Enforce client interface on the hub

Chapter 3

1. **A**. Microsoft.AspNetCore.SignalR.Client
2. **B**. Not automatically
3. **C**. `SendAsync` on .NET client and `invoke` on JavaScript client

Chapter 4

1. **D**. All of the above
2. **C**. Message type, the name of the event handlers and full message payload
3. **D**. Both of the above

Chapter 5

1. **C**. Context.ConnectionId
2. **C**. All of the above
3. **B**. When the client disconnects and reconnects

Chapter 6

1. **C**. All of the above
2. **A**. Send client stream to a relevant hub endpoint
3. **B**. Trigger a relevant hub endpoint and subscribe to it

Chapter 7

1. **B**. In a development environment
2. **C**. In either the server or the client configuration
3. **C**. All of the above

Chapter 8

1. **B**. CORS configuration
2. **D**. A system where authentication is shared between multiple applications
3. **C**. Authentication tells you who the user is, while authorization tells you whether the user has sufficient privileges to perform a particular action

Chapter 9

1. **B**. Running multiple copies of a particular server resource, so the load is evenly distributed between connected clients
2. **A**. A mechanism that allows multiple instances of a Hub to communicate with each other and act as a single Hub
3. **C**. An object that is positioned outside any SignalR Hub that can instruct the Hub to send messages to particular clients

Chapter 10

1. **D**. A service hosted in Azure that is used for scaling SignalR hub
2. **C**. Yes, but only via a special NuGet package and additional middleware configuration
3. **B**. Via REST API